DIGITAL SYSTEMS LABORATORY

STANFORD ELECTRONICS LABORATORIES
DEPARTMENT OF ELECTRICAL ENGINEERING
STANFORD UNIVERSITY . STANFORD, CA 94305

VERIFYING CONCURRENT PROGRAMS WITH SHARED DATA CLASSES

Susan Owicki

Technical Report No. 147

August 1977

This work was supported by the
Joint Services Electronics Program
under contract N-000 14-75-3-060 1.

VERIFYING CONCURRENT PROGRAMS WITH SHARED DATA CLASSES

by

Susan Owicki

Technical Report No. 147

August 1977

Digital Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

To be published in Proceedings of the Working Conference on Formal Description of Programming Concepts, August, 1977, North Holland

This work was supported by the Joint Services Electronics Program under contract N-00014-75-C-0601.

Digital Systems Laboratory

Departments of Electrical Engineering and Computer Science

Stanford University

Stanford, California 94305

Technical Report No. 147

August 1977

VERIFYING CONCURRENT PROGRAMS WITH SHARED DATA CLASSES

by

Susan Owicki

ABSTRACT

Monitors are a valuable tool for organizing operations on shared data in concurrent programs. In some cases, however, the mutually exclusive procedure calls provided by monitors are overly restrictive. Such applications can be programmed using shared classes, which do not enforce mutual exclusion. This paper presents a method of verifying parallel programs containing shared classes. One first proves that each class procedure performs correctly when executed by itself, then shows that simultaneous execution of other class procedures can not interfere with its correct operation. Once a class has been verified, calls to its procedures may be treated as uninterruptible actions; this simplifies the proof of higher-level program components. Proof rules for classes and procedure calls are given in Hoare's axiomatic style. Several examples are verified, including two versions of the readers and writers problem and a dynamic resource allocator.

Index terms: program verification, program proving, concurrency, parallel programs, monitors, classes, operating system design, shared classes

INTRODUCTION

Verifying concurrent programs is complicated by the nondeterministic way in which parallel processes can affect each other through operations on shared variables. Several language features for governing this interaction have been suggested: the monitor, (Hoare, 1974), (Brinch Hansen, 1973), is a particularly valuable tool. A monitor resembles a Simula class: it is a set of variables together with procedures that operate on those variables. Monitors have three restrictions not found on classes: 1) each variable in a concurrent program must belong to a monitor or be local to a process; 2) monitor variables may not be used outside the monitor; 3) the monitor includes scheduling operations that prevent two processes from executing monitor procedures at the same time. These restrictions provide a syntactic guarantee that two processes can not simultaneously operate on the same variable. Such a guarantee greatly simplifies the proof process. However, there are cases in which requirement 3 is overly restrictive. For example,

1. Some operations on shared variables can overlap in time without interference. Examples are read operations and operations which modify different parts of a shared data structure. Even operations which modify the same variables may do so in a way that is safe, despite simultaneous execution.
2. Objects which are syntactically shared may in fact be dynamically allocated to one process at a time, so that there is no possibility of interference.
3. Even in cases where each procedure requires mutual exclusion, it may be useful to allow the programmer to implement his own scheduling policy, rather than binding him to the monitor's standard policy.

Often a shared object will have some operations that fall in each of the categories above.

This paper discusses the specification and verification of shared data classes, a generalization of monitors in which mutual exclusion is not automaticallyprovided.

To verify a class, one must show that the procedures operate correctly (according to their specifications) even when executed in parallel. The verification of a process which uses the class can then be based on the specifications of the procedures, without regard to the details of their implementation.

It should be emphasized that the intent is not to advocate replacement of monitors by shared classes. The built-in mutual exclusion in monitors is a valuable aid to producing correct programs, and it greatly simplifies correctness proofs. Shared classes should be used only when they yield significant improvement in performance. In such cases, extra care must be devoted to insuring that the class works correctly no matter how its actions overlap in time; the proof techniques presented here. provide a means of verifying that it does.

The paper is organized as follows. Section 2 reviews the axiomatic proof method and provides a basis for the rest of the paper. The syntax of shared classes is described in Section 3, and proof rules for verifying the partial correctness of classes and processes are given in Sections 4 and 5. Section 6 contains a number of examples. Proofs of termination are discussed in Section 7. Finally, Section 8 summarizes and evaluates the results.

2. VERIFYING CONCURRENT PROGRAMS

The verification techniques in this paper are based on Hoare's axiomatic system for proving sequential programs (Hoare, 1969) and on a method for verifying parallel programs developed independently by Lamport (1977) and the author (Owicki and Gries, 1976). A brief review is given here.

Hoare's method is concerned with partial correctness. The partial correctness of a statement S is expressed by the formula $\{P\}S\{Q\}$, where P and Q are assertions. $\{P\}S\{Q\}$ implies that if execution of S begins with P true, then it either ends with Q true or never ends at all. Hoare provides a set of axioms and inference rules for proving partial correctness. For example,

<u>assignment</u> $\{P^x_E\}x := E\{P\}$, where P^x_E is the assertion formed by replacing every free occurrence of x in P by E

<u>consequence</u> $\dfrac{\{P'\}S\{Q'\}, P \vdash P', Q' \vdash Q,}{\{P\}S\{Q\}}$ where $P \vdash Q$ means that Q can be proved using P as an assumpt'on, and the notation $\frac{a}{b}$ means that b can be deduced if a has been proved.

A program proof can be presented informally by giving a proof outline, in which assertions are displayed at appropriate points in the program. This style is used in the examples in this paper. In a proof outline, each statement S is always directly preceded by one assertion, called its pre-condition or pre(S). Suppose $\{P\}S\{Q\}$ has been proved, and T is a statement in S. Then, if execution of S begins with P true, pre(T) holds whenever T begins execution.

Concurrent execution is initiated by a statement with the form

<u>cobegin</u> S_1 // S_2 // . . . // S_n <u>coend</u>.

The statements $S_1,\ldots,S_n$ are called parallel processes. There are no restrictions on the way in which parallel execution is implemented; in particular, nothing is assumed about the relative speeds of the processes. It is required, however, that each assignment statement or expression evaluation represent a single uninterruptible action. This rule can be relaxed when no ambiguity results. For example, the statement $x := x + 1$ can be treated as uninterruptible if x is a local process variable, but not if x is shared between processes. Actions which may safely be treated as uninterruptible are called <u>elementary</u>. The proof rule for shared classes in Section 4 gives conditions under which a procedure call may be treated as an elementary action, even though it is in fact an interruptible sequence of actions on shared variables.

The proof rule for cobegin statements is

$$\text{cobegin}\quad \frac{\{P_1\}S_1\{Q_1\},\ldots,\{P_n\}S_n\{Q_n\} \text{ are interference-free}}{\{P_1 \wedge \ldots \wedge P_n\}\ \underline{\text{cobegin}}\ S_1 // \ldots // S_n\ \underline{\text{coend}}\ \{Q_1 \wedge \ldots \wedge Q_n\}}$$

The rule states that the effect of executing $S_1,\ldots,S_n$ in parallel is the same as executing each one by itself, provided the processes do not "interfere" with one another. To show that they do not, one must prove that certain key assertions in the proof of $\{P_i\}S_i\{Q_i\}$ remain true under parallel execution of the other processes. In this case, the proof of $\{P_i\}S_i\{Q_i\}$ will still hold, and Q_i will be true on termination of S_i if P_i was true on initiation. For example, the assertion $(a \le b)$ remains true under execution of $b := b+1$, while the assertion $(b= 0)$ does not. The invariance of an assertion P under execution of a statement S is expressed by the formula $\{P \wedge pre(S)\}S\{P\}$. This is the basis of the interference-free property defined below.

Definition. Given a proof $\{P\}S\{Q\}$ and a statement T with precondition $pre(T)$, T does not interfere with $\{P\}S\{Q\}$ if the following formulas can be proved.

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

b) Let S' be any statement in S except a component of an elementary action.
Then $\{pre(S') \wedge pre(T)\}T\{pre(S')\}$.

Definition. $\{P_1\}S_1\{Q_1\},\ldots,\{P_n\}S_n\{Q_n\}$ are interference-free iff for each elementary action T in S_i, T does not interfere with $\{P_j\}S_j\{Q_j\}$ for $j \neq i$. (Of course, local variables of S_i may be renamed to avoid conflict with variables of S_j.)

It can be proved that when the cobegin rule is used, any computation that begins in a state with $P_1 \wedge \ldots \wedge P_n$ satisfied has the property that $pre(S)$ holds whenever S is ready to execute. This is because the sequential proof of the process containing S guarantees that $pre(S)$ will hold at the instant S becomes ready to execute, and the interference-free test insures that it will remain true in spite of the actions of other processes.

At times it is necessary to add statements containing auxiliary or ghost variables in order to verify a cobegin statement. The auxiliary variable should be used only in assignments to each other, so that their presence does not affect the program's control flow or its effect on non-auxiliary variables.

The use of the cobegin rule and auxiliary variables is illustrated by verifying the program

$$\text{Add2: } \underline{\text{cobegin}}\ x := x + 1 // x := x + 1\ \underline{\text{coend}},$$

under the assumption that $x := x + 1$ is an uninterruptible operation. Proving $\{x = 0\}\text{Add2}\{x = 2\}$ requires the addition of an auxiliary variable y. A proof outline for the augmented program is given below.

```
{x = 0}
y := 0;
{x = y ∧ y = 0}
cobegin
    {x = y} x := x + 1 {x = y + 1}
//
    {y = 0} [x := x + 1; y := 1] {y = 1}
coend
{x = y + 1 ∧ y = 1}
{x = 2}
```

Here the notation [x := x + 1; y := 1] indicates an elementary action; assignments to auxiliary variables can always be included in an elementary action becausethey take no "real" execution time. The sequential proof of each process is trivial. The interference-free test requires four steps, which show that each assertion in one process remains true under the action in the other process. For example, to show that $(x = y)$ is invariant under [x := x + 1; y := 1], one must prove

$$\{x = y \wedge y = 0\}\ x := x + 1;\ y := 1\ \{x = y\},$$

which is trivial.

3. SYNTAX OF SHARED CLASSES

A shared class closely resembles a monitor. It defines a set of variables, procedures which operate on the variables, and an initialization statement. The class variables may not be referenced outside the class itself. Execution of class procedures by different processes may overlap in time: this is the major difference between classes and monitors.

As an example, consider the class type Counter defined by the declaration below.

```
type Counter:class;
    begin var c:integer;mutex:semaphore;
        procedure Add(y:integer);
            begin var t:integer;
                wait(mutex); t:= x; x := t+y; signal(mutex)end;
        procedure Sub(y:integer);
            begin var t:integer;
                wait(mutex); t:= x; x :=t-y;signal(mutex)end;
        begin x := 0; mutex := 1 end
    end Counter
```

An instance of the class is created by the declaration

```
var C:Counter,
```

and the class procedures are then invoked by the statements C.Add(i) and C.Sub(i).

A few comments on the example are in order. The class procedures are written in terms of elementary actions; thus incrementing x requires two steps. Semaphores are used for synchronization here and throughout the paper, but the proof techniques to be developed are independent of the particular synchronization method. Note that Counter is essentially a monitor, since each procedure uses mutex to obtain exclusive access to the shared variable x. Later examples will illustrate other kinds of classes.

Several additional syntax restrictions are necessary for the proof methods presented later. First, each variable used inside a cobegin statement must be declared inside a class or a process. This simplifies the interference-free test, since a process variable can not be modified by an action of another process, and a class variable can only be changed in a class procedure. For similar reasons, a shared class variable may not be passed as an actual parameter to a procedure in another class, since this would complicate the interference-free test for that class. In addition, the actual var parameters in a procedure call must be distinct from each other as well as from the value parameters; this restriction is needed for Hoare's procedure call rule (Hoare, 1971).

Lastly, it must be possible to partition the class variables into control and data variables. The control variables are used for synchronization, and are invisible outside the class in the sense that they do not appear in its specifications. Each class procedure body has the form

begin declarations;enter;operate;exit end,

where the enter, exit and operate sections satisfy the following conditions.

1. enter and exit are elementary actions, and operate is composed of elementary actions (a procedure call to a previously-verified class is elementary).
2. enter and exit do not use (read or write) any data variables, and they call only locally-defined classes.
3. operate does not use any control variables.

In the class Counter, x is a data variable and mutex is a control variable; the enter and exit actions are wait(mutex) and signal(mutex).

Condition 1 prohibits class procedures from calling each other. Nonrecursive calls could be handled by in-line expansions of the procedure. Allowing recursive calls is possible, but would require a stronger proof rule than the one in Section 5.

Any procedure trivially satisfies conditions 2 and 3, since enter and exit can be null statements. However, class verification is usually impossible unless enter includes all the synchronization required to lock out procedures which could interfere with the operate section. Thus conditions 2 and 3 effectively prevent the operate section from containing a monitor-like wait which releases the mutual exclusion lock. In most monitors, wait and signal occur at the beginning and end of procedures, so this requirement is not too restrictive.

4. CLASS SPECIFICATIONS

Class specifications describe the visible characteristics of a class, including the initial values of data variables and the effect of each procedure on data variables and var parameters. A class invariant, which gives the relation between control and data variables, is used in proofs only inside the class itself. The components of the specifications for a class C are listed below; the program variables which may appear free in each assertion are indicated in parentheses.

C.Initial (C.data)
C.I(C.data,C.control) (the class invariant)
For each procedure C.p(var x;y)
  C.p.Pre(C.data,x,y,caller)
  C.p.Post(C.data,x,y,caller)
  C.p.Change(C.data) (the set of data variables changed by C.p),
    where caller is the identity of the process
    which invoked C.p, implicitly passed as a value parameter.

The specifications for class Counter of Section 3 are

Counter.Initial: $x = 0$
Counter.I: $0 \leq \text{mutex} \leq 1$
Counter.Add.Pre: $x = x_0$ Counter.Add.Post: $x = x_0 + y$
Counter.Add.Change: $\{x\}$
Counter.Sub.Pre: $x = x_0$ Counter.Sub.Post: $x = x_0 - y$
Counter.Sub.Change: $\{x\}$

Verification of class specifications is accomplished using the two-step method described in Section 2. The restrictions on variables in the clauses of the specifications limit the scope of the interference-free test in the proof.

Sequential correctness

1. {true} initialization {C.Initial $\wedge$ C.I}
2. For each procedure C.p,

   {C.p.Pre $\wedge$ C.I} body of p {C.p.Post $\wedge$ C.I}

   where for each statement S in the body of p, pre(S) $\vdash$ C.I.
3. For each procedure C.p, the set C.p.Change contains all data variables that appear on the left-hand-side of assignments in C.p, and all variables in D.q.Change for each procedure D.q. called in the operate part of C.p.

Interference

4. For each pair of procedures C.p and C.q, including p=q, the proofs in 2 above are interference-free, except possibly for the initial and final assertions C.p.Pre $\wedge$ I and C.p.Post $\wedge$ I.
5. If C contains a call to a class D not local to C, then each assertion P in the proof of C is invariant over actions of D, i.e. for all D.r

   {D.r.Pre $\wedge$ P} D.r(var a; e) {P}

Verifying the specifications in this way assures the C.I holds at all times (except possibly during elementary actions, which effectively take no time). In addition, if C.p.Pre is true at the instant when C.p.enter begins execution, it can be shown that C.p.Post must hold at the instant (if any) when C.p.exit terminates. When C.p executes by itself, the sequential correctness proof implies that it performs correctly. The interference-free tests cover every action which could interfere with C.p by modifying a shared variable. All shared variables belong to a class and can only be modified in class procedures. Step 4 in the proof checks all actions in class C and (implicitly) all classes local to C. Since the specifications of C contain only local variables, global variables can appear in assertions in C only as a result of a class call. Thus step 5 checks all global classes whose variables could appear in the proof of C. Because C.p.Pre and C.p.Post do not have to pass the interference-free test, it is necessary to assume that C.p.Pre holds at the instant when execution begins, and C.p.Post is not guaranteed to hold beyond the instant of termination. This will be discussed further in Section 5.

To illustrate the proof method, the class Counter is verified below. An array of auxiliary variables

<u>var</u> m: <u>array</u> processId of 0..1

has been added to give a stronger invariant,

$$\text{Counter.I}: 0 \le \text{mutex} \le 1 \wedge \text{mutex} = 1 - \left(\sum_{\text{process Id}} m[i] \right)$$

Sequential correctness

1. {true} x := 0; mutex := 1; m := 0 {x = 0 $\wedge$ Counter.I}

2. $\{x = x_0 \wedge \text{Counter.I}\}$
   [wait(mutex);m[caller] :=1]
   $\{x = x_0 \wedge m[\text{caller}] = 1 \wedge \text{Counter.I}\}$
   t := x;
   $\{t = x_0 \wedge m[\text{caller}] = 1 \wedge \text{Counter.I}\}$
   x := t + y;
   $\{x = x_0 + y \wedge m[\text{caller}] = 1 \wedge \text{Counter.I}\}$
   [signal(mutex); m[caller] := 0]
   $\{x = x_0 + y \wedge \text{Counter.I}\}$

   A similar proof can be carried out for Counter.Sub.

3. Both procedures change x and no other data variables, so Add.Change and Sub.Change are correct.

Interference

4. The interference-free property for Counter comes from the mutual exclusion provided by the semaphore operations. In general, if statements S and S' are mutually exclusive, a proof outline can be found in which

$$\text{pre}(S) \wedge \text{pre}(S') \vdash \text{false}.$$

Then the invariance of pre(S') under S follows immediately, for the test reduces to

$$\{\text{false}\}\ S\ \{\text{pre}(S')\}$$

and {false}T{P} can be proved for any T and P. In the class Counter, for example, the assertion

$$P:\ x = x_0 \wedge m[\text{caller}]= 1 \wedge \text{Counter.I}$$

must be proved invariant under execution of x := t +y by another process. The formula to be proved is

$$\{P \wedge \text{pre}(x := t+y)\}\ x := t+y\ \{P\}.$$

Now renaming caller to caller' in P gives

$$P \wedge \text{pre}(x:= t+y) \vdash I \wedge m[\text{caller}] = 1 \wedge m[\text{caller}'] = 1$$

where caller $\neq$ caller'. Then

$$P \wedge \text{pre}(x := t+y) \vdash \text{mutex} \leq -1 \wedge \text{mutex} > 0$$
$$\vdash \text{false}$$

Thus the statements are mutually exclusive,andtheinterference-free property must hold. The pre- and post-conditions of Add and Sub, which are notprotected by mutual exclusion, are not required to pass the interference-free test.

5. No verification is required here, since Counter does not call any other classes.

5. CLASS PROCEDURE CALLS

Once a class has been verified, programs using the class may safely treat calls to class procedures as elementary actions. From this viewpoint, a call to procedure C.p is equivalent to a nondeterministic assignment statement thatgives new values to the variables in C.p.Change. The new values must satisfy C.p.Post, but otherwise are unconstrained. The obvious proof rule for such an action is

$$\{\text{C.p.Pre}\}\ \text{C.p}\ \{\text{C.p.Post}\}.$$

However, this rule is not adequate. The proof of a process that calls C.p must pass the interference test, so the pre-condition of C.p (like all pre-conditions) must be invariant under the execution of other processes. Frequently the assertion C.p.Pre will not be acceptable. The proof rule below, a combination of Hoare's procedure call rule and rule of adaptation (Hoare, 1971), can be used to obtain a pre-condition which is interference-free.

Class procedure call: Let C be a class with specifications as described in Section 4. Then in process i

$$\{\exists \bar{k}(C.p.Pre' \land \forall \bar{b}(C.p.Post' \supset Q))\}\ C.p(\bar{a};\bar{e})\ \{Q\}$$

where $\bar{a}$ = actual var parameters

$\bar{e}$ = actual value parameters

$C.p.Pre' = C.p.Pre^{\bar{x}\ \bar{y}\ caller}_{\bar{a}\ \bar{e}\ i}$

$C.p.Post' = C.p.Post^{\bar{x}\ \bar{y}\ caller}_{\bar{a}\ \bar{e}\ i}$

Q is any assertion

$\bar{k}$ = variables free in C.p.Pre' or C.p.Post', but not in $\bar{a}$, $\bar{e}$, Q, or the class variables of C

$\bar{b} = \bar{a} \cup C.p.Change$

The rule of adaptation in sequential programs is discussed in Guttag, et al.(1977). For concurrent programs, it can be justified as a derived proof rule. For if {P}S{Q} can be proved using the class call rule, it can also be proved by expanding procedure calls in-line and using the techniques of Section 2. Details are given in the Appendix.

In most cases, the formulas proved using the class call rule can also be derived informally by treating the procedure call as a nondeterministric assignment. This view is legitimate for partial correctness, but not for termination. An assignment statement always terminates, but class procedures may loop or become blocked at a synchronization operation. Section 7 discusses methods of proving termination for class procedures and parallel programs.

As an example of the use of the class call rule, consider a proof of

$$\{x = t\}\ Counter.Add(1)\ \{x = t + 1\}.$$

Application of the call rule gives

$$\{\exists x_0(x = x_0 \land \forall x(x = x_0+1 \supset x = t+1))\}\ Counter.Add(1)\{x = t + 1\}.$$

Since $(x=t) \vdash (\exists x_0(x = x_0 \land \forall x(x = x_0+1 \supset x = t+1)))$, the rule of consequence yields

$$\{x = t\}\ Counter.Add(1)\ \{x = t + 1\}.$$

The class call rule allows the original pre- and post-conditions, $x = x_0$ and $x = x_0 + 1$, to be "adapted" to x=t and x=t+1. Thus the value of x can be related to a program variable rather than the artificial constant x_0. More important, the new pre-condition may be invariant under the actions of other processes where the old was not. This is illustrated in the proof outline below (the program is essentially the example of Section 2).

```
{x = 0}
t := 0;
{x = t ∧ t = 0}
cobegin
    {x = t} Counter.Add(1) {x = t + 1}
//
    {t = 0}[Counter.Add(1); t :=1] {t = 1}
coend
{x = 2}
```

The formulas for both calls to Counter.Add(1) are proved using the class call rule, as are the four interference tests. For example, to prove that x=t is invariant under [Counter.Add(1);t := 1], one must prove

$$\{x = t \land t = 0\}[Counter.Add(1);\ t := 1]\ \{x = t\},$$

which requires another use of the class call rule.

6. EXAMPLES

This section presents several examples of shared classes and discusses the main points in their verification.

Example 1. Monitor-like classes where all procedures are mutually exclusive are proved by first verifying the enter and exit actions. Oncemutual exclusion is established, the interference test is trivial. In Section 4, the class Counter was verified using this method. The most common reason for using a monitor-like class, rather than a monitor, is to provide a special-purpose scheduler.

Hoare's proof rules for monitors include an invariant J; J holds when no processes are executing monitor procedures. Such a monitor invariant is a special case of the class invariant. For example, if mutual exclusion is accomplished with a semaphore mutex, and J is a monitor invariant, then $(mutex = 1 \supset J)$ is an equivalent class invariant. Hoare's proof rules for monitorscan bederived from the class rules, but only for monitor procedures which fit the enter;operate;exit pattern.

Example 2. A class for managing a dynamically allocated resource needsprocedures for acquiring and releasing a resource unit, as well as the usual procedures for operating on it. A process first executes Acquire, then some sequence of operations, then Release. Acquire and Release require mutual exclusion, but resource operations do not; they have null enter and exit actions. The declaration below shows the important features of such a class.

```
type Alloc: class;
    begin  var  Resource unitId of resource;
                 : powersetoof unitId;
             freeCount,mutex: semaphore;
             owner: array unitId of processId;
             a,b. auxiliary integer;

        procedure Acquire(var unit: unitId);
            begin [wait(freeCount); a :=a+l]
                wait(mutex);
                unit:= oneof(free); owner[free] :=caller;
                [free := free-{unit};  a:=a-1]
                signal(mutex)
            end

        procedure Release(unit: uni tId);
            begin if owner[unit] ≠ caller then return;
                wait(mutex);
                [free := free ∪ {unit}; b := b + 1]
                owner[free] := null;
                signal (mutex);
                [signal(freeCount); b := b-l]
            end

        procedure opl(unit: unitId,...);
            begin operate on Resource[unit] end

        procedure op2...

        begin free := allUnits; freeCount:= unitCount;
            owner := null; a := 0; b := 0
        end

    end Alloc
```

The control variables of Alloc are mutex and freeCount; the other variables are data variables. Alloc is specified as follows:

Initial: free= allUnits $\wedge$ owner= null $\wedge$ a = b= 0

I: ($\forall i \in$ unitId($i \in$ free $\supset$ owner[i]= null) $\wedge$ freeCount= size(free)+ b-a
$\wedge$ $0 \leq$ freeCount $\leq$ unitCount)

Acquire.Pre: true
Acquire.Change: {free,owner,a}

Acquire Post: owner[unit]= caller

Release.Pre: owner[unit]= caller
Release.Change: {free,owner,b}

Release.Post: unit$\in$free

op1.Pre: owner[unit]= caller $\wedge$...
op1.Change: {Resource[unit]}

op1.Post: owner[unit] = caller $\wedge$...

op2.Pre: owner[unit] = caller $\wedge$ . . .
op2.Change: {Resource[unit]}

op2.Post: owner[unit] = caller $\wedge$...

The sequential step in the class verification is quite straightforward. The interference-free proof involves four cases.

1. {Acquire,Release} under {Acquire,Release}: mutual exclusion provided by mutex.
2. {Acquire,Release} under {op1,op2}: op1 and op2 do not modify any variables needed in assertions of Acquire or Release.
3. {op1,op2} under {Acquire,Release}: the only variable used in op1 or op2 and modified in Acquire or Release is owner[unit]. The pre-condition of ach action in op1 or op2 should include owner[unit] = caller. Now Acquire only changes owner[i] when owner[i]= null; Release only changes owner[i] when owner[i] = caller', where caller $\neq$ caller'. Thus neither will change owner[unit] while op1 or op2 is being executed.
4. {op1,op2} under {op1,op2}: Suppose op1 and op2 are executed at the same time. Let owner[unit]=caller in op1 and owner[unit'] = caller' in op2; then caller $\neq$ caller'. Thus unit $\neq$ unit', and the two procedures are operating on different resource units.

Example 3. Many search table organizations allow searches to go on in parallel with each other and with the addition of a new entry. The specifications of such a table manager are given below. The table stores keys in a data structure T. The form of T is not important, but the entries are indexed in some way, and select(T,i) retrieves the entry with index i. Maxsize gives the maximum number of entries in T, and size(T) gives the current number.

```
type table: class
    begin var T: tabletype;

    Initial : Size(T) = 0
    I: $size(T) \leq Maxsize \wedge \forall i,j(Select(T,i) = Select(T,j) \supset i = j)$

    procedure Insert(x: key;var i: Tindex);
        Insert.Pre: $T = T_0 \wedge size(T) < Maxsize \wedge \forall i(select(T,i) \neq x)$
        Insert.Post: $extend(T,T_0,x) \wedge select(T,i) = x$, where
            $extend(T,T_0,x) \equiv \exists i(select(T,i) = x \wedge select(T_0,i) = null$
                $\wedge \forall j(i \neq j \supset select(T,j) = select(T_0,j)))$
        Insert.Change: {T}

    procedure Search(x: key;var i: Tindex);
        Search.Pre: $T = T_0$
        Search.Post: $(i \neq null \supset Select(T,i) = x) \wedge$
                $(i = null \supset \forall j(Select(T_0,j) \neq x))$
        Search.Change= $\phi$
```

The specifications for Search imply that x will be found if it was in T when Search began. If it was added later, it may or may not be found.

The code for procedures Search and Enter is not given; presumably sequential verification is possible. For parallel execution, Search can not interfere with either Search or Insert, since it does not modify any shared variables. In most cases, parallel Inserts would not be safe, so Insert must include code to lock out other Inserts. To avoid interference with Search, Insert should be written in such a way that the assertion $(T = T_0 \vee \text{extend}(T, T_0, x))$ holds after each action; this will guarantee that Search and Enter are interference-free.

Example 4. Lamport (1977) presents an interesting pair of database operations to show that elementary actions (to use the terminology of this paper) are not necessarily sequential. In his example, operations p and q givedifferent (correct) results depending on whether execution of p precedes, follows, or overlaps execution of q. Thus if p and q occur in parallel, the resulting state is different from one which could be reached by p;q or q;p. It is still possible, however, to specify and verify the correct behavior of p and q as elementary actions.

Example 5. The well-known readers and writers problem involves a file which must be synchronized so that read operations can occur in parallel, but a write operation blocks both reads and writes. Several solutions have been published, e.g. Courtois, et al. (1971), Brinch Hansen (1972). Here we show how such a file can be represented and specified as a class.

```
type rwfile: class;
    begin var f: array 1..filesize of frecord;
              reading: auxiliary array processId of 0..1;
              writing: auxiliary array processId of 0..1;
              ...synchronization variables...
     Initial: $\forall i \in$ processId(reading[i]=writing[i]= 0)
     I: $(\max(\text{reading}[i]) = 0 \vee \Sigma\ \text{writing}[i] = 0) \wedge \Sigma \text{writing}[i] \le 1$
    procedure startread;
         startread.Pre: reading[caller] = 0
         startread.Post: reading[caller] = 1
         startread.Change: {reading[caller]}
    procedure endread;
         endread.Pre: reading[caller] = 1
         endread.Post: reading[caller]= 0
         endread.Change: {reading[caller]}
    procedure startwrite;
         startwrite.Pre: writing[caller]= 0
         startwrite.Post: writing[caller] = 1
         startwrite.Change: {writing[caller]}
    procedure endwrite;
         endwrite.Pre: writing[caller] = 1
         endwrite.Post: writing[caller]= 0
         endwrite.Change: {writing[caller]}
    procedure read(i: 1..filesize; var x: frecord);
         begin x := f[i] end
         read.Pre: reading[cailer]= 1          read.Post: x = f[i]
         read.Change: $\phi$
    procedure write: (i: 1..filesize; x: frecord);
         begin f[i] := x end
         write.Pre: writing[caller] = 1        write.Post: f[i] = x
         write.Change: {f[i]}
    begin reading := 0; writing :=0 end
end rwfile
```

Procedures startread   endwrite can be taken from any of the solutions to the problem. Their correctness is assumed. Sequential proofs for read and write are trivial. For the interference-free proof, one must first show that startread ... endwrite can not interfere with read by changing reading[caller]. This is easy, since startread and endread only change reading[caller'], where caller $\neq$ caller', and startwrite and endwrite do not change reading at all. The proof for write is similar. For the various pairs of read, write operations, the interference tests are passed as follows.

read/read: no shared variable modified in read

read/write: mutual exclusion is implied by the invariant
$(\max(\text{reading}[i]) = 0 \vee \Sigma\,\text{writing}[i] = 0)$

write/write: mutual exclusion is implied by the invariant
$(\Sigma\,\text{writing}[i] \leq 1)$

Example 6. The class defined above has one drawback. It is possible for a process to use the class unsafely - for example, by calling read without having called startread. Such a program could not be verified, since the pre-condition of read requires reading[caller]=1. It would be better, however, if unsafe actions could be prevented altogether. The class pFile implements a protected file.

```
type pFile: class
    begin uFile: rwfile;
        Initial: true
        I: $\forall$ i $\varepsilon$ processId (i $\neq$ caller in any procedure $\supset$
                              reading[i] = writing[i] = 0)
        procedure pRead(i: 1..filesize; var x: frecord);
            begin uFile.startread;uFile.read(i,x);uFile.endread end
            pRead.Pre: true
            pRead.Post: x=uFile.f[i]
            pRead.Change = $\phi$
        procedure pWrite(i: 1..filesize; x: frecord);
            begin uFile.startwrite;uFile.write(i,x);uFile.endwrite end
            pWrite.Pre: true
            pWrite.Post: uFile.f[i] = x
            pWrite.Change = {uFile.f[i]}
    end pFile
```

The class uFile is declared inside pFile; thus its procedures are not accessible outside of pFile. Since uFile has already been verified, calls to its procedures can be treated as elementary actions while verifying pFile. Thus the enter and exit actions are elementary, as required. Given the specifications of uFile, class pFile is easy to verify.

7. TERMINATION

Section 5 showed how class procedures can be treated as nondeterministic assignment statements in partial-correctness proofs. This is not valid for termination proofs, however. For example, the procedure defined by

```
procedure stop;begin wait(s); signal(s) end
```

is equivalent to a null statement with respect to partial correctness, but not with respect to termination. To deal with termination, the class specifications must be extended to include delay assertions giving the conditions under which each procedure may fail to terminate. To prove termination for a process that calls a class procedure, one must prove that the procedure's delay condition can not remain true.

When the cobegin statement was introduced, no assumptions were made about the implementation of concurrency. To deal with termination, however, some knowledge of the scheduling rules is required. Here two assumptions are made. First, the scheduler is fair; i.e. a process proceeds at a non-zero rate unless it blocks, executing wait(semaphore). Second, if a process is blocked at wait(s), it must continue after a bounded number of signal(s) operations have been performed. Thus semaphore scheduling must be fair, although not necessarily first-come-first-served.

The proof methodology for termination is essentially that of Lamport (1977); it is reviewed informally here. The basic notion is a relation $A \rightarrow B$ (A leads to B) between sets of program states.

Definition. Let A and B be sets of states for a program S. Then $A \rightarrow B$ iff a computation which reaches a state in A must eventually reach a state in B.

Sets of program states will be described by assertions, which may include the predicates start(i,L) to indicate that process i is ready to executethe statement labeled L, and finish(i,L) to indicate that i has just finished statement L.

Formulas like $A \rightarrow B$ are proved by starting with a partial-correctness proof outline for the program S and applying axioms and inference rules for termination. For example,

assignment L: x := E
$$\frac{\text{legal}(E)}{(\text{start}(i,L) \wedge \text{pre}(L)) \rightarrow (\text{finish}(i,L) \wedge \text{post}(L))}$$

The rules for other sequential statements are similar and will not be given here. For concurrent statements,

cobegin L: cobegin $L_1 : S_1 // \ldots // L_n : S_n$ coend
$$\frac{\text{start}(i,L_i) \rightarrow \text{finish}(i,L_i), \text{ for } 1 \leq i \leq n}{(\text{start}(0,L) \wedge \text{pre}(L)) \rightarrow (\text{finish}(0,L) \wedge \text{post}(L))}$$

signal L: signal(sem)
$$(\text{start}(i,L) \wedge \text{pre}(L)) \rightarrow (\text{finish}(i,L) \wedge \text{post}(L) \wedge \text{sem} = \text{sem}' + 1)$$

In the axiom for signal, sem' represents the value of sem at the instant when signal began. It may not be possible to deduce sem= sem'+1 from the postcondition of L, since that assertion must be interference-free. However, the state sem= sem'+1 must occur, even though it may not persist, and this is reflected in the axiom.

The rule for wait(sem) is harder to express, since wait will terminate only if other processes execute a sufficient number of signals.

wait L: wait(sem)
$$\frac{(\text{start}(i,L) \wedge \text{pre}(L) \wedge \text{sem} = 0) \rightarrow \text{sem} > 0}{(\text{start}(i,L) \wedge \text{pre}(L)) \rightarrow (\text{finish}(i,L) \wedge \text{post}(i,L) \wedge \text{sem} = \text{sem}' - 1)}$$

In order to prove that L: wait(sem) terminates, one must show that the semaphore sem can never stay at zero while a process is waiting at L. In other words, other processes are guaranteed to perform enough signal operations to allow process i to pass L.

The termination rule for a class call involves its delay assertion. C.p.Delay is verified by proving

$$C.p.\text{Delay} \rightarrow \neg C.p.\text{Delay} \vdash \text{start(body of } p) \rightarrow \text{finish(body of } p),$$

In other words C.p. must terminate unless the condition C.p.Delay persists throughout execution of C.p. Once C.p.Delay has been verified, the following rule can be used to prove that a call to C.p. terminates.

<u>class procedure call</u> L: C.p(a;e)

$$\frac{(\text{start}(i,L) \wedge \text{pre}(L) \wedge C.p.\text{Delay}) \rightarrow \neg C.p.\text{Delay}}{(\text{start}(i,L) \wedge \text{pre}(L)) \rightarrow (\text{finish}(i,L) \wedge \text{post}(L) \wedge C.p.\text{post}')}$$

$$\text{where } C.p.\text{post}' = C.p.\text{post}\,{}^{\bar{x}}_{\bar{a}}\,{}^{\bar{y}}_{\bar{e}}\,{}^{\text{caller}}_{i}$$

The procedure call rule has the same form as the semaphore wait rule. In fact, wait and signal are essentially procedures in a pre-defined class, with

sem.wait.Delay:	sem=0
sem.signal.Delay:	false.

The examples below illustrate verification of the delay clause and its use to prove properties related to termination.

<u>Example 1.</u> In the class Counter of Section 3, both procedures are guaranteed to terminate, i.e.

Counter.Add.Delay:	false
Counter.Sub.Delay:	false

The delay clause is verified using the partial correctness proof outline in Section 4. Both Counter.Add and Counter.Sub have the form

```
{m[caller] = 0 ∧ Counter.I}
a: [wait(mutex);m[caller] := -1]
{m[caller] = -1 ∧ Counter.I}
sequential statements
b: [signal(mutex);m[caller]:= 0]
{m[caller] = 0 ∧ Counter.I}
```

where $\text{Counter.I} \supset ((\text{mutex} = 1 - \sum_i m[i]) \wedge 0 \leq \text{mutex} \leq 1)$

(The original proof outline did not include $m[\text{caller}] = 0$ in the pre-condition of the wait operation, but it is obviously valid, and could be derived formally by adding another auxiliary variable.)

To verify the delay clause, one must show

$$\text{false} \rightarrow \neg\,\text{false} \vdash \text{start(caller},a) \rightarrow \text{finish(caller},b)$$

or

$$\vdash \text{start(caller},a) \rightarrow \text{finish(caller},b).$$

The sequential operations will always terminate, as will b: signal(mutex), so it is only necessary to show that a: wait(mutex) terminates. From the proof rule for wait, wait(mutex) must terminate if

$$(\text{pre}(a) \wedge \text{mutex} = 0) \rightarrow \text{mutex} > 0$$

can be proved. Now

$$(\text{pre}(a) \wedge \text{mutex} = 0) \vdash (\text{Counter.I} \wedge m[\text{caller}] = 0 \wedge \text{mutex} = 0)$$
$$\vdash \exists \text{caller}'(m[\text{caller}'] \neq 0 \wedge \text{caller} \neq \text{caller}').$$

Thus some process caller' is in the sequential part of Counter.Add or Counter.Sub. Eventually caller' will execute V(mutex), leaving mutex > 0. Thus $(\text{pre}(a) \wedge \text{mutex} = 0) \rightarrow \text{mutex} > 0$, as required, and the delay clause is verified.

Example 2. For the dynamically allocated resource (Example 2 in Section 6), the delay clauses are

Alloc.Acquire.Delay: free= empty
Alloc.Release.Delay: false

Acquire can not be blocked permanently at wait(mutex) by the reasoning used above. It can be blocked forever at wait(freeCount) only if the condition freeCount= 0 remains true. Now the class invariant implies

$$freeCount = size(free)+b-a.$$

As long as freeCount= 0, no process can add to a, and a must eventually return to 0. Thus

$$freeCount = 0 \rightarrow (a = 0 \vee freeCount > 0).$$

Combining this with the invariant yields

$$free \neq empty \rightarrow freeCount > 0.$$

Thus

$$(free = empty \rightarrow free \neq empty) \vdash (start(wait(freeCount)) \rightarrow finish(wait(freeCount)))$$

and

$$(free=empty \rightarrow free \neq empty) \vdash start(Acquire) \rightarrow finish(Acquire),$$

ending the verification of the delay clause.

To verify that a call to Alloc.Acquire terminates, one must show that

$$free=empty \rightarrow free \neq empty$$

in the program which uses Alloc. This can usually be accomplished by showing that each process that acquires a resource unit will eventually release it. For example, suppose each parallel process has the form

<u>begin</u> S1; Alloc.Acquire; S2; Alloc.Release <u>end</u>

where S1 and S2 do not contain calls to Alloc.Acquire or Alloc.Release. If S2 can not be blocked,

$$finish(i,Alloc.Acquire) \rightarrow finish(i,Alloc.Release)$$

which implies that the set of free units can not remain empty forever. If S1 can not be blocked, the process must terminate, and if all processes have this form, the entire <u>cobegin</u> statement must terminate.

In many applications, concurrent programs are intended to run forever. Termination proofs are not relevant for such programs, but it is often important to show that a process can not be "starved," i.e. permanently blocked. For processes with the following form:

```
do forever
    L1: begin S1; Alloc.Acquire; S2; Alloc.Release; L2: end,
```

freedom from starvation can be expressed by

$$start(i,L1) \rightarrow finish(i,L2).$$

By the same reasoning as before,the processes can not starve if S1 and S2 can be proved to terminate.

Example 3. The first solution to the readers and writers problem (Courtois, et al., 1971) was unfair to writers in the sense that a stream of readers could keep a writer permanently locked out of the file. With this implementation, the delay clauses for the class rwfile include

rwfile.startread.Delay= false
rwfile.startwrite.Delay = max(reading[i]) = 1.

Proving that a process calling rwfile.startwrite can not starve requires proving that there will always be a time when no readers are using the file. Such a proof would be impossible for most programs. In general, the use of an unfair scheduling algorithm makes starvation a real possibility.

8. SUMMARY

Any proof method for concurrent programs must account for the many ways that actions from different processes can be interleaved during execution. Thetechniques described in Section 2 handle such interleaving by requiring that the assertions used in proving each process be unaffected by the actions of other processes. The proof rule for shared classes (Section 5) reduces the number of steps in this interference test; it treats a procedure call as an indivisible non-deterministic assignment statement. Section 4 gives the rules for verifying that partial correctness is preserved under this transformation. Although information about termination may be lost, it can be recovered from the delay clause in the procedure specifications. Treating procedure calls as elementary operations simplifies both partial-correctness and termination proofs by decreasing the number of distinct actions to be considered. Proving that a class meets its specifications can in principle be quite complex, because of the interference test. In most cases, however, verification is relatively simple because the procedures fall into one of the following categories.

1. Procedures provide code for mutual exclusion.
2. Procedures do not modify shared data.
3. Procedures operate on different parts of the shared data.
4. Shared data is dynamically allocated to one process at a time.

In these cases the proofs will be straightforward. It is only when procedures simultaneously operate on the same data that a detailed interference test will be required. This latter kind of programming is in general so difficult and unreliable that it should be avoided except under extreme efficiency constraints. Most practical applications of shared data classes fall into one or more of theclasses above and so are not hard to verify.

The proof techniques developed in this paper are similar in spirit to the reduction method (Lipton, 1974, 1976). Reduction also simplifies the proof of parallel programs by allowing a sequence of actions to be treated as a unit. The two methods differ in the kinds of sequences that can be grouped together, the means of proving that a grouping is safe, and the kinds of propertiesthatarepreserved. In reduction, the actions to be combined must have the same effect in all executions and must be guaranteed to terminate once started. Class procedures, on the other hand, may represent non-deterministic actions and may fail to complete. Reductions are justified by proving that once a sequence is started, it can not be blocked, even temporarily. Usually this is easier than proving partial correctness by the interference-free method. Reduction preserves both the values computed by a program and its termination/deadlock properties; information about process starvation may be lost. The class procedure rule preserves the values computed; termination information is lost, but can be recovered from the delay assertion. Partial correctness, termination, and safety from deadlock or starvation can be proved with the shared class techniques. Lipton presents reduction primarily as a tool for proving freedom from deadlock, but it could also be used for partial correctness and termination. Overall, the conditions for applying reduction are quite strict, and the class procedure rule can be used in many programs where reduction is not safe. The price of this flexibility is the potential complexity in proving that a class meets its specifications. When both methods apply, reduction is likely to give an easier proof.

Monitors are a special case of shared classes; their semantics are such that monitor procedures may always be considered elementary. Silberschatz, et al., (1377) propose extending concurrent Pascal (Brinch Hansen, 1975) with a new type, the dynamically-managed class. This is another instance in which easier proof rules apply. A fruitful extension of this work would be the application of the general proof rule to derive simpler rules for important special cases.

REFERENCES

Brinch Hansen, P. (1972). Acta Informatica, 1, 190.

Brinch Hansen, P. (1973). Operating Systems Principles. (Prentice Hall, Englewood Cliffs, New Jersey).

Brinch Hansen, P. (1975). IEEE Trans. on Software Eng., SE-1, No. 2, 199.

Courtois, P. J., Heymans, R., and Parnas, D.L. (1971). Comm. ACM, 14, No. 10, 667.

Engeler, E. (1971). Symp. on Semantics of Algorithmic Languages. (Springer-Verlag New York).

Guttag, J., Horning, J. and London, R. (1977). Proc. Formal Desc. of Programming Concepts, (North Holland, Amsterdam).

Hoare, C.A.R. (1969). Comm. ACM, 12, No. 10, 576.

Hoare, C.A.R. (1971). in Engeler, 102.

Hoare, C.A.R. (1974). Comm. ACM. 17, No. 10. 549.

Lamport, L. (1976). Mass. Computer Associates Report CA-7610-0712.

Lamport, L. (1977). IEEE Trans. on Software Eng., SE-3, No. 2, 125.

Lipton, R. J. (1974). Yale Computer Science Research Report #30.

Lipton, R. J. (1976). Comm. ACM, 18, No. 12, 717.

Owicki, S. and Gries, D. (1976) Acta Informatica 6, 319.

Owicki, S. (1977) Technical Report 133, Digital Systems Lab, Stanford.

Silberschatz, A., Kieburtz, R. B., Bernstein, A. J. (1977). IEEE Trans. on Software Eng., SE-3, 210.

APPENDIX

In this appendix, the class procedure call rule of Section 5 is justified by showing that it can be derived from the other axioms and inference rules.

Theorem. Suppose that C is a class in a program S, where procedures of C are called only from the processes of S and not from other classes. Let S' be the program obtained from S by replacing each call C.p($\bar{a}$;$\bar{e}$) of a class procedure C.p.(var $\bar{x}$,$\bar{y}$) by

begin $\bar{x}:=\bar{a}$; $\bar{y}:=\bar{e}$; body of C.p; $\bar{a}:=\bar{x}$ end.

Then if {P}S{Q} can be proved, {P}S'{Q} can also be proved.

Repeated application of this theorem can be used to remove all class calls, because there is no cycle of classes which call each other. (If S had such a cycle, {P}S{Q} could not be proved, since all classes called by C must be verified before C).

Before sketching a proof of the theorem, we review the structure of class procedures and define some names for assertions and sets of variables. Recall that the variables of C can be partitioned into data and control variables: let C.c name the set of control variables and C.d the data variables. Each procedure C.p has the form

begin declarations;enter;operate;exit end,

where enter and exit are elementary actions on C.c and operate is composed of elementary actions on C.d. A proof outline for C.p has the form

```
C.p.(var x̄,ȳ): {p.pre ∧ I}
               p.enter;
               {p.entered}
               p.operate;
               {p.leaving}
               p.exit;
               {p.post ∧ I}
```

where the program variables free in p.pre and p.post are from C.d., $\bar{x}$, or $\bar{y}$, and the other assertions may have free variables from C.c. in addition. We will require $p.pre \equiv \exists C.c(p.entered)$, i.e. p.pre is equivalent to p.entered with respect to variables in C.d. Such a choice for p.pre is legitimate for the proof outline, since p.enter isindependent of the variables in C.d. Likewise, we require $\exists C.d(p.entered) \equiv \exists C.d(P)$, for each assertion P from p.entered to p.leaving, i.e. p.entered and P agree on C.c. Again, this is legitimate, since p.operate is independent of C.c. Recall that each assertion in the proof outline, except the first and last, is invariant over all actions in class procedures.

In order to prove the theorem, we must show how to derive a proof for P{S'}Q, given one for P{S}Q. The approach will be to define a tranformation M on the assertions in a proof of S such that, for each statement T in S,

$$\{P\}T\{Q\} \vdash \{M(P)\}T'\{M(Q)\}.$$

A simple choice for M(P) is P itself. This would be adequate for the sequential proof, but in general it will not pass the interference test. This is because the grain of action in S is the class procedure, while in S' it is the elementary action within the procedure. Even though P was interference-free in S, it may not be in S'. Thus, M(P) must be a weaker assertion than P itself. Now if M(pre(S)) implies that the variables accessed by S are in a state consistent with pre(S), executing S will result in a state in which those variables are consistent with post(S), regardless of the values of other variables. The syntax of shared classes allows S' to be partitioned into three kinds of statements, which use disjoint sets of variables.

{T1}	main program	neither C.c nor C.d
{T2}	p.enter,p.exit	C.c.
{T3}	p.operate	C.d

(This is a slight oversimplification - the main program may call a class which is also called in C, so that its variables are in either C.c or C.d. Step 5 in the interference test for verifying C assures that this will do no harm).

For a statement T1 in the main program, M(pre(T1)) will imply that all non-class variables are in a state consistent with pre(T1), and that if all processes were to finish executing class procedures, pre(T1) would hold in the resulting state. The assertion WH(P,Active), defined below, essentially states that P would hold if all processes left the class. Let

Active= {v: process v is executing a class procedure in[v]};

(Active and in are auxiliary variables in S'). Then

$$WH(P,V) \equiv \forall a((\bigwedge_{v \in V} in[v].post \supset P)$$

where a = ∪(in[v].Change uvar parameters of in[v]).

Note that WH(pre(T1),Active) implies that non-class variables are in a state consistent with pre(T1), so T1 executes exactly as it would if pre(T1) were true. The only exception is at a procedure call T= $C.p(\bar{a},\bar{e})$ in S, which is started by p.enter in S'. In the proof of S, the class call rule was used to verify T, so

$$pre(T) \vdash \exists k(p.pre \wedge \forall \bar{a}(p.post \supset post(T)))$$

(The substitution of actual parameters is ignored here, since it is covered by assignment statements in S'. Also, we can add an auxiliary variable to record the values of k which satisfy the assertion; this makes it possible to remove $\exists k$ from the assertions.)

Now if WH(pre(T),Active) holds before class entry, then WH(post(T),Active) will hold after entry, because the process executing enter joins the active set. WH(post(T),Active) remains true throughout the class body, because all variables which can be modified by C.p are quantified in WH(post(T),Active). So just before p.exit, WH(post(T),Active) holds; also p.leaving holds (this will be proved shortly). Since p.leaving $\vdash$ p.post, and p.post $\wedge$ WH(post(T),Active) $\supset$ WH(post(T),Active $\sim$ {v}), WH(post(T),Active) must hold after exit. Thus, letting pre(T') $\equiv$ WH(pre(T),Active) is sequentially valid. It is also interference-free: the quantifiers prevent actions in the class procedures from affecting WH(pre(T),Active), and pre(T) is invariant over actions of other processes outside C.

For actions in procedure C.p to perform correctly, it is sufficient that C.p.pre hold on entry. If T is a call to C.p.($\bar{a}$,$\bar{e}$) in S, we know that WH(pre(T)) holds before entry, and pre(T) $\vdash$ p.pre. But we need to know that p.pre, not just WH(p.pre), holds after p.enter. This can be proved by adding the assertion $\bigwedge_{r \in \text{procedures}}$ WHB(pre(T),Active,r) to M(pre(T)). WHB(pre(T),Active,r) essentially states that if pre(T) $\vdash$ r.pre (which is always true when T is a call to C.r), then r.pre would hold if all procedures which can block r would finish.

To express WHB(pre(T),Active,r), we need to consider the circumstances under which procedure C.r can be entered. Since r.enter uses only control variables, C.r can be entered unless some C.q has set the control variables to lock out C.r. Procedures r and q are said to lock (lock(r,q)) if there is no control state consistent with both of them being in execution.

The proof of C showed that C.r and C.q are interference-free. Because there is no interaction between the enter/exit actions and the operate section, it can be proved that the non-interference comes about in one of two ways

1) lock(q,r): C.q and C.r can not be in execution at the same time
2) $\neg$lock(q,r): the non-interference test involved no knowledge of the state of control variables. Thus for T an action in p.q, if C.r does not interfere with pre(T) then C.r does not interfere with $\exists$C.c(pre(T)). In particular, C.r does notinterferewith p.pre.

Returning to the assertion WHB(pre(T),Active,r), if pre(T) $\vdash$ r.pre, then WHB(pre(T),r) should guaranteethat r.pre would holdifprocedure C.r wereentered. Now any procedure C.q which can falsify r.pre must satisfy lock(r.q), so C.r can not pass r.enter while C.q is in execution. Now WH(pre(T)) means that pre(T) would hold if all processes left class procedures. But r.pre would hold if all procedures which block r were inactive, since the other procedures do not falsify r.pre. This gives the formula for WHB(pre(T),Active,r):

$$\text{WHB(pre(T),Active,r)} \equiv \forall \text{ variables(pre(T)} \supset \text{r.pre)} \supset \text{WH(r.pre,rBlock)},$$
$$\text{where rBlock} = \text{Active} \cup \{q\text{: block(r,q)}\}$$

The final form of the transformation M(P), foranassertion P inprocess v, not in a class body, is

$$\text{WH(P,Active)} \wedge \forall\, r\ \text{(WHB(P,Active,r))} \wedge \text{C.I} \wedge \text{in[v]} = \text{null} \wedge v \notin \text{Active}$$

For an assertion P in the body of C.p in process v, where Q is the postcondition of the call of C.p, M(P) is

$$\text{WH(Q,Active)} \wedge \forall\, r\ \text{(WHB(Q,Active,r))} \wedge P \wedge \text{in[v]} = \text{C.p} \wedge v \in \text{Active}$$

A formal proof that M yields a valid proof for S' can be accomplished by using induction on the structure of statements T in S, to show that

$$\{pre(T)\}T\{post(T)\} \vdash \{M(pre(T))\}T'\{M(post(T))\}.$$

The proof is not given here; it primarily involves manipulation of logical formulae. Hopefully, the reader is satisfied with the informal arguments for sequential validity. Non-interference for WH has been discussed; non-interference for WHB is similar. The only difference is that actions in C.q with $\neg lock(r,q)$ can modify variables in r.pre without invalidating r.pre, so those variables do not have to be quantified in WHB. Finally, for T in a class procedure, pre(T) must be invariant over actions in C, and actions outside C can not affect its variables.