PROOF TECHNIQUES FOR RESOURCE CONTROLLER PROCESSES

Krithivasan Ramamritham

Department of Computer and Information Science
University of Massachusetts
Amherst MA 01003

June 1982
82-18

This material is based upon work supported by the National Science Foundation under grant MCS 82-02586.

ABSTRACT

Shared resources and the processes that control them play a critical role in the functioning of concurrent systems. A shared resource is viewed as an abstract data type consisting of the definition of the resource and the operations on it with additional synchronization constraints. An abstract model is introduced for expressing the behavior of resource controller processes. The model sheds light on various aspects of resource control that need to be specified. Based on this model, we present techniques for verifying resource controllers using the formalism of temporal logic. Properties of the operations on a given shared resource are first verified. This is followed by the verification of invariant and temporal properties of the controller. Using the verified properties of the resource controller, we prove the properties of the processes sharing the resources. The techniques are illustrated by their application to resource sharing tasks in Ada. As a prerequisite for achieving this, the semantics of Ada tasking primitives are defined along with associated proof rules.

Note for the Printer

The symbol ¥ stands for "for all" and should be printed as ∀.

The symbol ⊣ stands for "there exists" and should be printed as ∃.

The symbol <̵ stands for "belongs to" and should be printed as ∈.

The symbol [] should be printed as □.

The symbol <> should be printed as ◇.

1. INTRODUCTION

The problem of analyzing concurrent systems can be alleviated by viewing a system as being made up of a set of resources, and processes that use these resources. A shared resource would be managed by a controller process which is made responsible for the correct use of the resource. Our interest is in the verification of concurrent systems by first verifying the behavior of resource controller processes and then, using their specifications, verifying the behavior of the resource sharing processes.

A resource can be considered to be an abstract data type consisting of the resource definition and the access operations on the resource [6]. A shared resource has the added restriction that the operations be executed such that the shared resource is always in a consistent state. To ensure this, use of a shared resource is controlled by employing mechanisms such as monitors [8], serializers [2], sentinels [12] and the Ada tasking facility employing the rendezvous mechanism [4]. In this paper, we refer to such mechanisms as resource controllers. Given the crucial role played by resource controllers, it is essential that they be guaranteed to function correctly. This paper presents techniques for proving the correctness of resource controllers.

We accomplish this task by first introducing an abstract model for resource controller behavior. In this model an access operation goes through four phases: request, service, active and termination. The model takes into account the temporal ordering of the phases of operations when users make concurrent requests. It is also possible to model the execution context of a phase, i.e., whether the phase is executed by the process which makes the request, the resource controller, or some third process. We develop techniques for the verification of resource controllers in light of this model. Recognizing that a resource controller exercises all its control during the service phase of each operation, we translate the specifications of the invariant properties of the resource and its controller into constraints on the service of individual operations and then verify that the resource controller services each operation according to these constraints. Verification of fairness of the controller necessitates taking into account the control flow properties of resource controller code.

A number of methods have been proposed for proving properties of synchronization mechanisms [2, 9] but unified methods for handling both safety and liveness properties have started receiving attention only recently. To state the properties of a particular resource control mechanism and reason about the interactions of concurrent processes through resource sharing, properties of a resource controller throughout its execution should be specified. Thus standard axiomatic techniques will not suffice. Techniques based on temporal logic [17], on the other hand, lend themselves to specifying individual properties of interest without specifying the complete behavior. Using temporal logic Owicki and Lamport present a proof method for the liveness properties of concurrent programs synchronized through semaphores [16]. They acknowledge the need for further research in order to be able to handle more sophisticated language features. Our work is intended to be a step in that direction. Application of temporal logic to proofs of concurrent programs is also reported in [14] and [3]. While both utilize semaphores for synchronization, the former uses a programming language with assignments and GO TOs, whereas the latter utilizes a flowchart model of concurrent programs. Our techniques on the other hand, are applicable to programming languages such as Ada which have high-level synchronization primitives. They grew out of our efforts to prove properties of sentinel processes [19].

In this paper, we illustrate our proof techniques by applying it to resource control in the context of Ada. In Ada, tasks are the program units for concurrent programming. A shared resource manifests itself as an Ada package containing the resource being accessed, the operations on the resource and the controller. Our model paves the way for specifying the semantics of resource controller tasks in Ada. The formal denotational semantics of Ada given in [11] does not include the semantics of tasking primitives. Also, since we include fairness within the class of properties to be verified, the denotational approach will not be suitable [16]. Therefore, we provide a temporal semantics for Ada tasking primitives by specifying their behavior as predicate transformers as well as by specifying their liveness properties.

This paper is organized as follows. We start with a brief introduction to temporal logic in section two. The proposed model for shared resource access is explicated in Section three. In Section four, examination of Ada's rendezvous mechanism in the context of our model leads to the formal definition of the behavior of resource controller tasks in Ada. The proof technique is developed in Section five. (A case study involving the application of the proof technique is presented in the appendix.) Proof of a system of concurrent resource sharing processes is the subject of Section six. Section seven is devoted to concluding remarks.

2. TEMPORAL LOGIC

Pnueli first applied temporal logic for reasoning about invariant and time-dependent properties of concurrent programs [17, 18]. Following along those lines, concurrency is modeled by a nondeterministic interleaving of computations of individual processes. Each computation changes the system state which consists of values assigned to program variables and the instruction pointer of each of the processes. Using temporal logic operators, one can specify and reason about the properties of the sequence of states that results from the execution of the concurrent processes.

Since temporal logic is an extension of predicate calculus, a temporal logic statement can involve the usual logical operators V (or), & (and), ~ (not) and => (implication) besides the temporal operators [], <> and UNTIL. The operator [] is pronounced "always". []P states that P is true now and will remain true throughout the future. The operator <> is pronounced "eventually" and is the dual of [] in that

    <>P IFF ~[]~P.

Thus, <>P if P is true now or will be true sometime in the future. A requirement such as "every request will be serviced" can be specified as

    []{"request for service exists" => <>"request serviced"}.

The operator UNTIL has the following interpretation:

    (P UNTIL Q) IFF as long as Q is false, P will be true.

(The truth value of P once Q becomes true is not indicated by UNTIL.) The UNTIL operator is typically used for expressing temporal orderings. For example, the fact that a service can not be provided until there is a request for that service can be stated as

    ~("request serviced") UNTIL ("request for service exists")

In addition to these three primitive operators, we have a derived operator to state that a predicate P can become true only after predicate Q does:

(P ONLYAFTER Q) <=> (~P UNTIL Q).

The semantics of [] and <> are identical to those of the corresponding linear time logic operators of [13] whereas UNTIL is related to Lamport's binary [] operator (read AS LONG AS) in the following manner:

(A UNTIL B) $\underline{=}$ (~B[]A)

<u>Temporal Logic Theorems:</u>

Following are some of the theorems of temporal logic that will be employed in this sequel.

T1: []A => {A & <>A & <>[]A & [][]A}

T2: <>[]A => []<>A

T3: [](A V B) => ([]A V <>B)

T4: [](A => B) => {[](<>A => <>B & <>(A & B)) &
[](A => B UNTIL ~A)}

T5: []A & []B => [](A & B)

T6: {(A UNTIL B) & []~B} => []A

T7: {[A UNTIL (B&A)] & A & <>B} => <>(B & A)

T8: {[A ONLYAFTER B & B ONLYAFTER C] => [A ONLYAFTER C]}

T9: {[](A => (B UNTIL C)) & [](C => (B UNTIL D))}
=> [](A => B UNTIL D)

T10: {[](A => (B UNTIL C)) & (D ONLYAFTER A) &
(C ONLYAFTER (~C & D))} => [](D => B)

3. A MODEL FOR RESOURCE SHARING IN CONCURRENT SYSTEMS

A shared resource comprises of the following:

- the resource being shared,
- the operations used to access the resource, and
- the associated resource controller.

We refer to each distinct type of access operation as an operation class. Each instance of a class is referred to as an operation in that class. Thus, for example, two different Read accesses to a shared database will correspond to two distinct Read operations. All accesses to a shared resource are through the execution of one of the operations defined on the resource and occur when permitted by the controller for that resource.

Execution of an operation goes through the Request, Initiation, Active and Termination phases. The request phase for an operation begins after a resource controller recognizes that a user program requests the execution of that operation. The request phase ends when the controller's internal data structures reflect the fact that a request is waiting for service. The time at which the controller permits execution depends on the state of the shared resource, priority associated with the request, invariant properties of the resource, etc. These determine the necessary conditions for executing an operation. The service phase begins when and if the necessary conditions hold and the resource controller decides to permit the execution of the operation. In this manner, the resource controller guarantees that the specified invariant properties are maintained. At the end of the service phase, the resource controller's internal data structures reflect the fact that permission has been granted for the execution of the operation. Thus the term "service" is equivalent to "granting of permission". The _active phase_ begins after the service phase ends. It is in this phase that the resource access defined by the operation takes place. The active phase ends when access is complete. The _termination phase_ begins after the active phase ends. At the end of the termination phase the resource controller's internal data structures reflect the fact that the operation has completed execution.

Temporal ordering of the phases in an operation:

The notation

p¦op

will be used to refer to phase p of operation op. To precisely define the model, we introduce some additional notation. Since each phase is executed by some process, a phase can be associated with specific statements in the code for that process. Given a statement S that is executed by some process,

at(S)	IFF	control of that process is at the beginning of S, i.e., the process's instruction pointer points to S.
in(S)	IFF	control is within S, i.e., the process's instruction pointer is pointing to some component of S.
after(S)	IFF	control is immediately following S, i.e., the instruction pointer is pointing to the statement following S. Thus, given a statement sequence S;T, after(S) <=> at(T)

These three predicates are mutually exclusive and become true in the above order. The formal definition of the language construct corresponding to S would specify how its component statements are affected by the execution of S (see section four).

Now we define the temporal ordering of the phases. In what follows, the variable "a" is universally quantified over the set of all operations. The four phases associated with an operation "a" are totally ordered in time as follows:

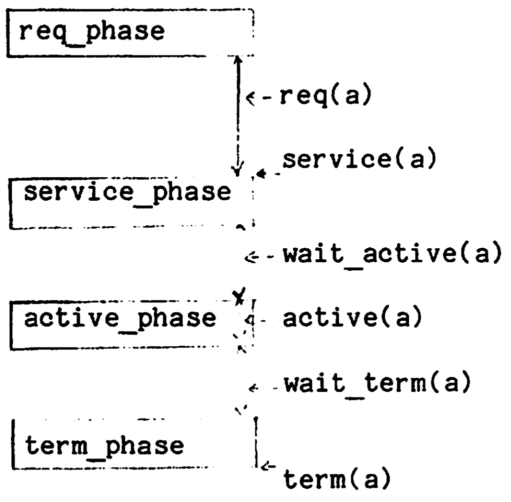
```
at(service_phase¦a)   ONLYAFTER after(req_phase¦a)
at(active_phase¦a) ONLYAFTER after(service_phase¦a)
at(term_phase¦a)   ONLYAFTER after(active_phase¦a)
```

The use of ONLYAFTER (instead of IFF) in the above statements reflects the possibility of delays between the execution of two consecutive phases. The above statements, in addition to the fact that

```
after(p¦a) ONLYAFTER at(p¦a)
```

for all phases p of operation a, define the sequential ordering of the phases of any operation a.

To precisely specify the state of each operation we introduce some predicates whose truth values are depicted in the following figure. Their formal definitions follow.



When a request is present for operation a, the predicate <u>req(a)</u> is true. A request for an operation can be said to be present only after the request phase for that operation begins.

```
     req(a) ONLYAFTER at(req_phase¦a)
```

req(a) is true at the end of the request phase and remains true until after the service phase has begun.

```
     after(req_phase¦a) => {req(a) UNTIL [at(service_phase¦a) & req(a)]}
```

A request ceases to exist at the end of the service phase.

```
     after(service_phase¦a) => []~req(a)
```

Thus it is not possible to determine the truth value of req(a) when control is within the request phase or the service phase since requests come into existence during the request phase and cease to exist during the service phase.

The predicate <u>service(a)</u> is true if and only if control is at the beginning of the service phase for a.

```
      service(a) <=> at(service_phase¦a)
```

Also, service(a) cannot be true unless the appropriate request is present.

```
      service(a) => req(a)
```

When a serviced operation is waiting to become active, wait_active(a) is true.

```
        wait_active(a) ONLYAFTER after(service_phase¦a)

        after(service_phase¦a)
          => {wait_active(a) UNTIL [at(active_phase¦a) & wait_active(a)]}

        [in(active_phase¦a) V after(active_phase¦a)] => []~wait_active(a)
```

The predicate active(a) is true when the access actually takes place. Therefore,

active(a) ONLYAFTER at(active_phase¦a)

at(active_phase¦a) => [active(a) UNTIL after(active_phase¦a)]

after(active_phase¦a) => []~active(a)

The predicate wait_term(a) is true when access a is complete and termination phase has not begun. Its definition is similar to that of wait_active(a).

The predicate term(a) is true if and only if control is at the end of the termination phase.

term(a) <=> after(term_phase¦a)

We use the model for specifying and verifying the behavior of resource controller processes. To do so, it is necessary to delimit the phases of access operations as well as define the relationship between the phases of different operations. Delimiting the phases involves identifying when at(p¦a) and after(p¦a) hold for each phase p of every access operation a. That in turn will define when the predicates associated with an operation are true. Now we introduce some terminology which will aid in defining the relationship between phases, and between phases and processes that execute the phases.

Execution Context of the phases: In order to specify the processes that execute the phases of an operation, we make use of the following function:

execution_context: SOC x SPH -> SPR

where

SOC = Set of Operation Classes,

```
SPH = {req_phase, service_phase, active_phase, term_phase},
SPR = {requesting_process, controller_process, temporary_process}.

execution_context(OPC,p) = pr
 if and only if
  phase p of operations in class OPC are executed by process pr.
```

Ordering relationships between phases:

A resource controller is a sequential process in that it can take only one step at a time. Hence from the beginning to the end of execution of phase p1 executed by a controller, control cannot reach the beginning of any other phase p2 executed by the same controller.

at(p1) => ∀p2≠p1, at(p2) ONLYAFTER after(p1)

We then say that p1 is uninterruptible. The notion of uninterruptibility is essentially an abstraction and is included because there is no logical necessity for a controller to start executing a new phase before a previous one is completed. Thus, at any given time, the controller can be in at most one phase of some unique operation. We identify this operation by specifying what the current operation is. For any phase p executed by a controller, the value of current_operation is defined as follows:

```
{at(p) & current_operation=c}
 =>
{current_operation=c UNTIL [current_operation=c & after(p)]}
```

4. SEMANTICS OF RESOURCE CONTROLLER TASKS IN ADA

In Ada, tasks are the program units for concurrent programming. We define the resource being accessed, the operations on it and the controller task within a package. An entry definition within the task can be thought of as an operation on the resource. A call on an entry within a task can be executed only when there is a Ready ACCEPT statement corresponding to that entry. The call is ACCEPTed when a rendezvous occurs. A rendezvous consists of executing statements between a DO and an END following the ACCEPT statement. Thus it is during a rendezvous for an entry that the corresponding access operation is executed. A condition can be associated with each ACCEPT statement. These can be viewed as the conditions that a controller imposes on servicing an operation. To reason about resource controllers, we formally specify the semantics of Ada tasking constructs, namely Entry calls, ACCEPT and SELECT statements. For further deatils of Ada's tasking facility, the reader is referred to [4, 10].

To illustrate tasking in Ada and to serve as a running example, the following resource control problem is introduced.

> A set of processes communicate through a Single Slot Buffer (SSB). A message is Deposited in the buffer by some process, Read by others, and Removed by some process. A new message cannot be deposited unless the previous message has been removed. Obviously, Deposit and Remove operations should exclude each other as well as Read operations. To keep the contents of the buffer current, Deposit and Remove operations have priority over Read operations.

Now we specify formally the properties of the Single Slot Buffer, its controller, and operations on the buffer. A variable "status" is used to denote whether the buffer is full or empty. Its value is modified by Deposit and Remove operations. Our fairness requirements are that if the buffer is full repeatedly, that is infinitely often, then a Remove operation should be eventually serviced. A similar fairness is required for Deposit operations. Due to the priority for Deposit and Remove operations over Read, Read requests are not expected to be serviced with fairness.

```
SPECIFICATION OF RESOURCE STATE CHANGE
 ∀d∈Deposit, []{after(term_phase|d) => status=full}
 ∀m∈Remove, []{after(term_phase|m) => status=empty}

SPECIFICATION OF CONSTRAINTS ON RESOURCE ACCESS
 ∀d∈Deposit, ∀m∈Remove, ∀r∈Read,
  [](service(d) => status=empty)
  [](service(m) => status=full)
  [](service(r) => status=full)

SPECIFICATION OF MUTUAL EXCLUSION OF OPERATIONS
 ∀d1,d2∈Deposit, d1≠d2,  []~{active(d1) & active(d2)}
 ∀m1,m2∈Remove,  m1≠m2,  []~{active(m1) & active(m2)}
 ∀m∈Remove, ∀d∈Deposit, []~{active(m) & active(d)}
 ∀d∈Deposit, ∀r∈Read,   []~{active(d) & active(r)}
 ∀m∈Remove, ∀r∈Read,    []~{active(m) & active(r)}

SPECIFICATION OF INVARIANT PROPERTY OF THE RESOURCE
 [] (status=full V status=empty)

SPECIFICATION OF PRIORITY
 ∀r∈Read, ∀m∈Remove, ∀d∈Deposit,
 []{[req(r) & req(d)] => service(r) ONLYAFTER service(d)}
 []{[req(r) & req(m)] => service(r) ONLYAFTER service(m)}

SPECIFICATION OF FAIRNESS
 ∀m∈Remove, ∀d∈Deposit,
  {req(m) & []<>(status=full)} => <>service(m)
  {req(d) & []<>(status=empty)} => <>service(d)
```

Ada code for the SSB problem is given on the following page. A call on the procedure Deposit (Remove) translates into a call on the entry corresponding to Deposit (Remove) whereas in order to allow for concurrent Reads, a call on the procedure Read translates into two entry calls with the actual access occurring between the calls. (This is necessitated by the restrictions placed on the specifications of entries in Ada.)

```
PACKAGE ssb IS
  PROCEDURE Read (m: OUT INTEGER);
  PROCEDURE Deposit (m:  IN INTEGER);
  PROCEDURE Remove;
END ssb;
PACKAGE BODY ssb IS
TASK ssb_controller IS
 buffercontents : INTEGER;
 ENTRY start_read;
 ENTRY end_read;
 ENTRY Deposit (m:  IN INTEGER);
 ENTRY Remove;
END ssb_controller;

TASK BODY ssb_controller IS
 TYPE state IS (full, empty);
 status : state := empty;
 #active_reads : INTEGER := 0;
 BEGIN
LL:  LOOP
LS:    SELECT
            WHEN Deposit'COUNT=0 & Remove'COUNT=0
                   & status=full =>
LRS1:        ACCEPT start_read
LRS2:        DO #active_reads := #active_reads+1; END
       OR
            WHEN true =>
LRT1:        ACCEPT end_read
LRT2:        DO #active_reads := #active_reads-1; END
       OR
            WHEN #active_reads=0 & status=empty =>
LDS:         ACCEPT ~~write~~(m:  IN INTEGER)
             DO     ~~Remove~~ Deposit
LDA:          buffercontents := m;
             END;
LDT:         status := full;
       OR
            WHEN #active_reads=0 & status=full =>
LMS:         ACCEPT Remove;
LMT:         status := empty;
   END SELECT;
  END LOOP;
 END ssb_controller;
PROCEDURE Read (m: OUT INTEGER) IS
     BEGIN
      ssb_controller.start_read;
 LRA:  m:=buffercontents;
      ssb_controller.end_read;
     END;
PROCEDURE Deposit(m:  IN INTEGER) IS
 BEGIN  ssb_controller.Deposit (m) END;
PROCEDURE Remove IS
 BEGIN  ssb_controller.Remove END;
END ssb;
```

Henceforth we will say that "a Deposit (Remove) operation is executed through an entry call" and that "a Read operation is executed through a procedure call". To keep track of concurrent Read operations, a counter, #active_reads, is utilized. The attribute "count" of an entry is used to determine the number of waiting entry calls.

In Ada, the request phase is kept hidden from the user, i.e., in the code for a controller task there are no statements corresponding to the request phase. The other phases, however, are present in the code. For example, statements LDS, LDA and LDT correspond to service, activation and termination phases of Deposit operations. (Later in this section, we will be discussing how statement-phase correspondences can be made.)

4.1 Semantics of Language Constructs for Tasking in Ada

Sequential Programming Constructs

We will be utilizing the following sequential programming constructs: assignment, LOOP and BEGIN..END statements. For an assignment statement

```
L:   X:=e;
```

```
at(L) => {at(L) UNTIL after(L)}
{at(L) & P(X,e)}  => {<>after(L) & [](after(L) => P)}
```

where P(X,e) is a predicate derived by substituting expression e for every occurrence of variable X in predicate P. Thus an assignment statement has no control points within it and hence in(L) is never true. (A null statement also has no control points within it.)

```
L: BEGIN
L1: <executable statement>;
L2: <executable statement>;
    ....
Ln: <executable statement>
   END
```

$at(L) \Leftrightarrow at(L1)$

$in(L) \Leftrightarrow \{\exists j\ 1 \le j \le n,\ in(Lj)\ \vee\ \exists j\ 2 \le j \le n,\ at(Lj)\}$

$after(L) \Leftrightarrow after(Ln)$

$at(L) \Rightarrow \{at(L)\ \mathrm{UNTIL}\ (in(L) \vee after(L))\}\ \&\ \langle\rangle\{in(L) \vee after(L)\}$

$in(L) \Rightarrow \{in(L)\ \mathrm{UNTIL}\ after(L)\}$

$\forall i\ 1 \le i < n,\ after(Li) \Leftrightarrow at(Li+1)$

$\forall i,j,\ j \ne i,\ [\,]\sim\{[at(Li) \vee in(Li)]\ \&\ [at(Lj) \vee in(Lj)]\}$

The last two statements indicate the sequential execution of statements within a BEGIN..END block. Semantics of the DO..END construct is the same as the BEGIN..END construct. Semantics of the LOOP..END construct is the same as the BEGIN..END construct with the following addition:

$after(L) \Rightarrow at(L)$

Concurrent Programming Constructs

There are three basic language constructs for tasking in Ada: ACCEPT and SELECT statements, and entry calls. An ACCEPT statement is executed in response to an entry call, at which time a rendezvous between the called task and the calling task occurs. A SELECT statement allows for a nondeterministic selection from a number of possible executions of ACCEPT statements. We first consider ACCEPT and SELECT statements. Entry calls are considered later in the context of calls on operations.

ACCEPT statement

An ACCEPT statement takes the following form:

```
L1: ACCEPT A
L2: DO  <executable statement> END;
```

When control reaches L1, it remains there until an entry call on A occurs, after which the first element in the queue for A is removed and L2 is executed. Thus, the behavior of the Accept statement is as follows:

```
A1: at(L1) => [at(L1) UNTIL (at(L1) & A'COUNT>0)]

A2: {C & at(L1) & A_q'first=a & A_q'rest=Q1}
     =>
    <>{C & after(L1) & A_q=Q1)}

A3: at(L1) => {at(L1) UNTIL after(L1)}

A4: after(L1) <=> at(L2)
```

where Q1 belongs to the domain of queues, a is an instance of type A, A_q is the queue allocated for entry A, A_q'first refers to the first element in A_q and A_q'rest refers to A_q without the first element. C is an arbitrary predicate that does not refer to any queue. A2 states that the truth value of such a predicate is not modified by an ACCEPT statement.

SELECT statement The SELECT construct allows for a combination of waiting for and selection from one or more alternatives. Selection can be controlled by conditions associated with each alternative of the selective wait statement. We now present the semantics for the basic form of the SELECT statement.

```
L:     SELECT
        WHEN G1 =>
 L11:    ACCEPT A1
 L12:      DO  <executable statement> END;
 L13:      <executable statement>;
       OR
       ....
       OR
        WHEN Gm =>
 Lm1:    ACCEPT Am
 Lm2:      DO  <executable statement> END;
 Lm3:      <executable statement>;
       END SELECT;
```

(For any j, Lj2, Lj3 or both could be null.) When control reaches a

SELECT statement, all guards {Gi, i=1..m} are evaluated to determine which alternatives are candidates for selection. These are known as <u>open alternatives</u>. If Gj is true, Lj1 may be executed if an entry call has been made on Aj. If several entries have been called, one of the open alternatives is selected arbitrarily.

The guards associated with the ACCEPT statements are boolean expressions. We distinguish between two types of guards: a guard is <u>internal</u> to a task if its truth value can change only due to some activity within the task. Otherwise, it is termed <u>external</u>. For instance, if a guard involves a global variable it would be external, in which case, it could become false soon after it is evaluated to be true. In this sequel, we confine our attention to guards that do not involve global variables. Even in such a case, the truth value of a guard could change after it is evaluated. Suppose for some i, one of the terms in Gi is Aj'COUNT=c for some j and c. We say that Gi "depends on a specific request count". It is possible that when Gi is evaluated, Aj'COUNT=c and Gi evaluates to true. However an entry call on Aj could be made after Gi is evaluated thereby making Gi false. In such cases, we derive Gi', called the "count_free_component" of Gi, by substituting "true" for every occurrence of Aj'COUNT=c in Gi.

Now we formally define the behavior of the above SELECT statement. Let N stand for the set of natural numbers starting from 1 and let M stand for the set {1,2,..,m} where m is the number of SELECT alternatives. Once control is at the beginning of the SELECT statement, guards G1 to Gm are evaluated to determine the open alternatives, at which time the predicate OAD, which stands for Open_Alternatives_Determined, becomes true. The set open_alternatives (closed_alternatives) consists of names of the entries corresponding to the true (false) guards.

S1: If control is at the beginning of a SELECT statement, and if none of the guards are true, then the predicate SELECT_ERROR becomes true.

```
[at(L) & ∀j∈M ~Gj & ~OAD] => SELECT_ERROR
```

S2: Otherwise, the open alternatives are determined.

```
[at(L) & ∃j∈M, Gj & ~OAD]
 =>
[at(L) & OAD & ∀j∈M, {[Gj => Aj∈open_alternatives] &
                      [~Gj => A̅j∈closed_alternatives]}]
```

S3: Selection is postponed until one of the entries for the open alternatives has been called.

```
{at(L) & OAD}
 =>
{[at(L) & OAD] UNTIL ∃Aj∈open_alternatives, Aj'COUNT>0}
```

S4: Once at least one of the entries for open alternatives has been called, control will reach the beginning of an ACCEPT statement for one such entry. The other open alternatives are not executed.

```
{{at(L) & OAD & ∃Aj∈open_alternatives, Aj'COUNT>0}
  =>
 ∃Aj∈open_alternatives,
 {<>at(Lj1) & []{at(Lj1 => (~OAD & Gj' & Aj'COUNT>0)}
   & ∀k∈M, k≠j, ~at(Lk1) UNTIL after(L)}
```

Notice that the count_free_component of the guard of the selected alternative is true at the beginning of the first statement of that alternative. In general, if condition C holds when at(L), then the count_free_component of C will hold at the beginning of the selected alternative.

S5: Control is after the SELECT statement when execution of one of the SELECT alternatives is completed.

```
after(L) <=> ∃j∈M after(Lj3)
```

S6: The following expresses the sequential nature of the SELECT statement.

```
∀j∈M, ∀S∈{Lj1, Lj2, Lj3},
 []~{at(L) & [at(S) V in(S) V after(S)]}
```

S7: The following statement specifies what it means for control to be within a SELECT statement.

```
in(L) <=> ∃j∈M, [at(Lj1) V at(Lj2) V in(Lj2) V at(Lj3) V in(Lj3)]
```

S8: Inference Rule for the SELECT Statement

From the above semantics, we can derive the following inference rule for SELECT statements. For a cyclic SELECT statement with label L, that is, for a SELECT statement within a loop

```
[]∃k∈M, Gk  &  ∀j∈M, []{at(Lj1) => <>after(Lj3)}
=>
[]<>{at(L)  & ~OAD}
```

The hypothesis states that one of the guards is always true and that once an alternative is chosen, the statement corresponding to that alternative will terminate. Thus control will always return to the beginning of the LOOP.

To accommodate for the possibility of absence of open alternatives and entry calls, ADA provides an ELSE clause within a SELECT statement. Its semantics can be specified in a similar manner. In addition to the ACCEPT statement, Ada allows DELAY and TERMINATE statements as alternatives within a SELECT statement. Also, ADA permits timed entry calls whereby an entry call is made only if it can be accepted within a specified period. We do not consider these possibilities in this paper.

4.2 Semantics of Operations on Shared Resources

Corresponding to each operation on a shared resource, we can define an entry within the resource controller task. Each operation on a shared resource can then be executed by the controller task through an entry call which would result in a rendezvous. Since Ada has a built-in exclusion mechanism for performing a rendezvous, all operations would then be executed in exclusion. Thus to permit concurrent executions of operations in a class, for example Read operations in the SSB problem, the code for that class should be programmed as a procedure with appropriate synchronizing entry calls before and after the actual operation.

Semantics of operations executed concurrently

To be concrete, let us assume a class of operations C. If operations in class C can execute concurrently, then the code for C would be implemented as a procedure and will have the following form.

```
   PROCEDURE C
   BEGIN
   start_C;
LA:  <code for C>
   stop_C;
   END;
```

Within the task body, the code for entries start_C and stop_C will be defined as follows:

```
LS1: ACCEPT start_C
LS2:  DO  <modify internal variables to reflect service> END;

LT1: ACCEPT stop_C
LT2:  DO  <modify internal variables to reflect termination> END;
```

What constitutes the service, active and termination phases is specified below. Recall that the request phase is not visible in Ada resource controllers. If start_C_q (stop_C_q) is the queue for waiting calls on entry start_C (stop_C), then,

```
{at(LS1) & start_C_q'first=c} <=>
 {at(service_phase|c) & current_operation=c}

{current_operation=c & [at(LS2) V in(LS2)]} <=>
 in(service_phase|c)

{current_operation=c & after(LS2)} <=> after(service_phase|c)

after(service_phase|c) => <>at(LA)

at(LA) <=> ∃c∈C, at(active_phase|c)

after(LA) <=> ∃c∈C, after(active_phase|c)

{at(LT1) & stop_C_q'first=c} <=>
 {at(term_phase|c) & current_operation=c}

{current_operation=c & [at(LT2) V in(LT2)]} <=> in(term_phase|c)

{current_operation=c & after(LT2)} <=> after(term_phase|c)
```

These define the phases of an operation implemented as a procedure. The execution context of these phases is specified by the following statements.

```
execution_context(C,req_phase)=controller_process
execution_context(C,service_phase)=controller_process
execution_context(C,active_phase)=calling_process
execution_context(C,term_phase)=controller_process
```

Hence the request, service and termination phases of operations executed as procedures cannot be interrupted.

Semantics of operations executed in exclusion

Now we consider the case when an operation is executed through an entry call. In this case, the code for the operation takes the following form:

```
          LS1: ACCEPT C
          DO
     LS2:  <modify internal variables to reflect service>;
     LA:    <perform operation>;
          END;
     LT:  <modify internal variables to reflect termination>;
```

Here LS1 and LS2 correspond to the service phase of operations in C, whereas LA and LT correspond to active and termination phases respectively. These can be formally defined as before. The execution context of all these phases is the controller process and hence their executions cannot be interrupted. This case essentially differs from the previous one in the uninterruptibility of the active phase which in turn leads to the mutual exclusion of execution of operations.

4.3 Semantics of Queues

As mentioned earlier, Ada tasks utilize queues for waiting entry calls. We will first consider the queues necessary for operations in class C executed through procedure calls.

Q1: The request phase for an operation in C ends when the queue of the entry start_C has an element corresponding to that request. During the service phase of the operation, this element is removed from the queue. Req(a) is true if a request is present for operation a, i.e., there is an element in start_C_q. Hence we have the following equivalences:

$\forall c \in C$, {req(c) <=> $\exists n \in N$, start_C_q[n]=c

$\exists n \in N, \exists c \in C$, start_C_q[n]=c <=> start_C'COUNT>0

Similarly, a queue named stop_C_q is utilized for operations in C waiting to be terminated. Q[i] is the i-th element from the front of the queue.

Q2: The queues in Ada are FIFO. Hence operations in C are serviced (terminated) in the order in which calls to start_C (stop_C) enter the queues. Thus

```
∀a,b∈C, ∀i,j∈N [start_C_q[i]=a & start_C_q[j]=b & i<j]
               =>
               [at(service_phase¦b) ONLYAFTER at(service_phase¦a)]

∀a,b∈C, ∀i,j∈N [stop_C_q[i]=a & stop_C_q[j]=b & i<j]
               =>
               [at(term_phase¦b) ONLYAFTER at(term_phase¦a)]
```

Now we consider operations executed through entry calls. In this case, one queue is associated with each operation and the following are true.

<u>Q3</u>:
```
∀c∈C, {req(c) <=> ∃i∈N, C_q[i]=c}
  ∃i∈N,∃c∈C, C_q[i]=c  <=>  C'COUNT>0
```

<u>Q4</u>:
```
∀a,b∈C, ∀i,j∈N [C_q[i]=a & C_q[j]=b & i<j]
               =>
               [at(service_phase¦b) ONLYAFTER at(service_phase¦a)]
```

4.4 <u>Semantics of Counters</u>

The internal data structure for keeping track of concurrent active operations is a counter. Typically one counter exists per operation class. A counter is incremented by one during the service phase and is decremented by one in the termination phase.

<u>C1</u>: A counter always has a zero or positive value.

```
[](#active_C ≥ 0)
```

<u>C2</u>: When an operation in a class is waiting for activation, active, or waiting for termination, the counter for that class has a positive value.

```
∃c∈C, {wait_active(c) V active(c) V wait_term(c)} => #active_C>0
```

<u>C3</u>: More precisely, a counter of operations has a positive value only if some operation in that class is in its service, active or termination phases.

$\#active\_C > 0 \Rightarrow \exists c \in C,\ [in(service\_phase|c) \lor wait\_active(c) \lor active(c) \lor wait\_term(c) \lor in(term\_phase|c)]$

Note that counters are used only for operations that can be executed concurrently.

4.5 Semantics of Calls on Operations

A task accessing a resource through an entry call waits until the rendezvous is complete, after which it proceeds with the execution of the statement following the call. Hence the calling task can execute concurrently with the termination phase of the called operation. Thus, if R is the label of the entry call on C,

```
CT1: at(R) => ∃c∈C, {[<>req(c) & (at(R) UNTIL service(c))] &
                      after(R) ONLYAFTER after(active_phase|c) &
                      []{after(active_phase|c) => after(R)} }
```

On the other hand, the following is true when the access is through a procedure call.

```
CT2: at(R) => ∃c∈C, {[<>req(c) & (at(R) UNTIL service(c))] &
                      after(R) ONLYAFTER after(term_phase|c) &
                      []{after(term_phase|c) => after(R)}
```

Here the statement following the call can execute only after the termination phase is completed.

5. VERIFICATION OF RESOURCE CONTROLLER TASKS

Our motivation behind proving resource controller processes is to be able to prove properties of concurrent systems that use shared resources. In this regard, we show that given the specifications for a shared resource and its controller, it is possible to verify properties of a set of processes sharing the resource. This scheme will succeed if and only if the code for the resource controller meets the specifications.

Verification of a resource controller is performed in three steps. Step one involves showing that the code for each operation performs the changes to the resource as specified. Step two concerns invariant properties whereas step three concerns temporal properties. Now we give the details of these steps. In the appendix we apply this proof technique to the Single Slot Buffer Problem.

Step-1:

Since this step focusses on individual operations only, reasoning here is confined to sequential pieces of code. Hence standard axiomatic techniques suffice [5,7].

Step-2:

We saw earlier that a resource controller decides when an access operation should be serviced and that once an operation has been serviced, its active and termination phases follow. Thus any control over resource access should be exercised at the time of service. In other words, prior to service of an operation, it should be ascertained that the execution of the operation will maintain the invariant properties specified. This is carried out in step two via three substeps.

Substep-2.1: Here we derive the constraints imposed by the resource controller on servicing requests for access. These constraints arise due to the uninterruptible nature of phases executed by the controller task as well as due to the explicit constraints that appear in the WHEN clauses of the SELECT statement.

Due to our stipulation that any phase executed by the controller is uninterruptible, if phase p is being executed by a controller, the controller cannot be executing any other phase. Thus we have the following inference rule.

Inference-Rule-I

```
∀p1,p2, p1≠p2,
 {execution_context(p1)=controller_process &
  execution_context(p2)=controller_process}
  =>
 {[at(p1) V in(p1)] => [~at(p2) & ~in(p2)]}
```

For instance, since the execution context of the active phase of Deposit operations and the service phase of Read operations is the controller process, using the above inference rule and the semantics of the predicate "active" we have,

```
 ∀d∈Deposit, active(d)
                  =>
                  [at(active_phase|d) V in(active_phase|d)]
                  =>
                  ∀r∈Read,  [~at(service_phase|r)] => ~service(r)
```

and so,

```
∀r∈Read, ∀d∈Deposit, []{service(r) => ~active(d)}.
```

This constrains read operations to be serviced only when no deposit operations are active. Constraints resulting from the application of this rule along with those explicitly stated in the WHEN clauses, result in the overall constraints imposed by the resource controller.

Substep-2.2: In this substep we derive constraints imposed by the specifications of mutual exclusion, priority and others that affect service. For this we use the following transformation rules:

Mutual exclusion Transformation Rule

This rule specifies sufficient conditions for servicing operations in order to satisfy mutual exclusion requirements.

```
∀op1,op2,
{ [](service(op1) =>  ~[active(op2) V wait_active(op2)]} &
  [](service(op2) =>  ~[active(op1) V wait_active(op1)]} }
 =>
  [] ~ {active(op1) & active(op2)}
```

Note that the consequent of this rule expresses the exclusion between op1 and op2. The correctness of this rule is shown in the appendix.

Priority Transformation Rule

This rule (also proved in the appendix) aids in the translation of priority specifications into necessary conditions. Accordingly, priority specifications will be satisfied if an operation is serviced only when no requests for operations of higher priority are present.

```
∀op1,op2,op1≠op2,
[]{service(op1) => ~req(op2)}
  =>
[]{[req(op1) & req(op2)] => [service(op1) ONLYAFTER service(op2)]}
```

Recall that the consequent of this rule expresses the priority for op2 over op1.

Resource State Invariance Transformation Rule

> With the invariance as the post condition, using the specification of changes to the resource state, derive the weakest precondition for operations in each operation class.

The weakest preconditions become the constraints for executing operations in the respective operation classes. It should be verified that operations which change the resource state exclude each other and that if an operation is serviced when the precondition is true, the invariance will hold during the active phase of the operation.

The constraints on servicing an operation is the conjunction of (1) the constraints derived by the application of the above rules, (2) the implicit requirement that

$\forall op\ []\{service(op) \Rightarrow req(op)\}$

and (3) the specifications of constraints on resource access.

Substep-2.3: Here it is verified that the conditions derived in substep 2.1 imply those derived in substep 2.2, thereby showing that the controller services requests so that the invariant specifications are maintained.

Step-3:

In general, at any given time, a number of operations may have satisfied their necessary conditions. However, since a resource controller task takes one step at a time, it has to choose one of these operations to be serviced next. We have to show that this choice is made with the specified fairness. This is the purpose of step three and requires the examination of control flow aspects of the code.

Fairness specifications, as we detailed earlier, express the requirement that requests be serviced if appropriate conditions hold. The resource controller should choose the next operation to be serviced in accordance with the fairness specifications. In ADA, this choice manifests itself as the choice made among the open alternatives within a SELECT statement. Recall that in ADA if more than one open alternative is eligible for service, this choice is made arbitrarily. Our strategy for proving fairness properties in spite of this arbitrariness in selection can be summarized as follows:

> Step-3.1: Using the predicates that appear in the guards of the SELECT loop, derive the set of disjoint conditions that can occur at the beginning of the SELECT statement.

Our proof of eventual service for a request is by contradiction, i.e., we assume that a request is never serviced and then show that this request has to be eventually serviced. For this purpose, the following steps are performed for each condition.

> Step-3.2: Determine the alternatives that can be open when that condition holds.
>
> Step-3.3: Show that there will be an entry call corresponding to at least one open alternative.
>
> Step-3.4: Consider each of the open alternatives. Show that the specified request will be serviced.

Thus, irrespective of the condition that holds and the selection made, the specific request will be serviced.

6. VERIFICATION OF RESOURCE SHARING PROCESSES

So far we have focused on the processes that control access to shared resources. Now we give an example of proving properties of the processes sharing the resources from the code for these processes and the specifications of the shared resources and their controllers. Typically, this involves utilizing the semantics of entry calls and the specifications of fairness.

Let us consider two processes that access the single slot buffer, a process that deposits messages and another that removes messages. Their code is as follows:

```
--Depositing process                  --Removing process

L1: LOOP                              L2: LOOP
    m:=<create next message>;         LM: SSB.Remove
LD: SSB.Deposit(m)                        END;
    END;
```

The processes that Read the contents of the buffer are not shown here. We would like to prove that every deposit and remove request is serviced.

[]{at(LD) => <>after(LD)}

[]{at(LM) => <>after(LM)}

It can be formally shown that

[]<> ∃d∈Deposit req(d).

[]<> ∃m∈Remove, req(m)

that is, each process generates requests repeatedly. Now we invoke the fairness specification of the controller to prove that every request will be serviced.

The following is a tautology by T3.

[]<>(status=empty) V ~[]<>(status=empty)

which by the specification of invariance of resource state and the duality of [] and <> is

[]<>(status=empty) V <>[](status=full)

Let us consider the second possibility first. Consider a remove request m. If <>[](status=full) then by T2 and the specification of fairness to remove operations

<>service(m)

and since remove operations terminate

<>term(m)

which by the specification of changes to resources results in

<>(status=empty).

Since remove requests occur repeatedly, that is, []<>req(m), the tautology above reduces to []<>(status=empty).

Now we prove that at(LD) => <>after(LD). Assume at(LD). Then $\exists d \in$Deposit, <>req(d). If []<>(status=empty), by the fairness specifications of the Deposit operations, <>service(d). By the liveness properties of the controller, <>(after(active_phase¦d)), which by the semantics of calls on operations implies <>after(LD). Thus

[]{at(LD) => <>after(LD)}

The corresponding statement for the removing process can be proven in the same manner. This example demonstrates that very often it suffices to focus on the resource controller process and then by using the specifications of this process alone properties of the processes sharing the resources can be proven.

7. CONCLUSIONS

This paper presents techniques for the verification of resource controller tasks in Ada. As the first step towards achieving this, we introduced a model for expressing the behavior of shared resource controllers. It recognizes the following:

- Every access operation is executed as a result of a request from a user process.
- After determining that the conditions are appropriate for an access to proceed, a controller services the access request.
- The code for an access operation is executed only after the controller has serviced the operation.
- On completion of an access operation, certain clean-up actions are required.

The model intuitively conforms with the notion of resource access. To be able to utilize the model for the purpose of specification and verification, we formalized it by introducing predicates, and related their truth values to the execution of the phases. The relationships between phases was specified through temporal logic statements.

We have found the model to be general enough to be applicable to monitors [8], Ada Tasks [4], serializers [2] and sentinels [12]. In this paper, we dealt with Ada tasks. These mechanisms differ from one another not in their adherence to the model but in the restrictions that they impose on the execution of the phases. Using our formalism it is possible to explicitly specify these restrictions. The specifications of these restrictions along with the specifications of the semantics of the language constructs for tasking resulted in the semantics of resource controller tasks in Ada. Given that the denotational semantics of Ada presented in [11] does not include semantics of tasking constructs, this is a significant step. In ADA the open alternatives are first determined and then the selection is made from the open alternatives, Hence the semantics of the SELECT statement is more involved than the guarded selection in CSP [1]. Here we have focused on the primary features in Ada for tasking. Later, features such as task priorities and DELAY statements will be brought under consideration.

Of the four phases that occur during the execution of an operation, a resource controller has explicit control over only the service phase. The specification for a resource control problem should state how such control should be exercised. Our techniques for the verification of resource controllers were developed in the context of these specifications. Both invariant properties such as exclusion, and temporal properties such as fairness were included in our analysis of resource control. Proof of properties of individual operations required only a sequential reasoning. Invariant properties of the resource and the controller were transformed into conditions that hold when operations are serviced, thus localizing the reasoning required. Verification of fairness properties necessitated arguments concerning flow of control during the selection of the next request to be serviced. Thus our proof techniques cover all properties of interest to resource control in a concurrent environment and are applicable to high-level resource control mechanisms.

Confining synchronization information for a given shared resource within a controller process makes it possible to first prove the properties of the controller process and then utilize its specifications for proving the properties of resource sharing processes. Thus, in such situations, proofs of "non-interference" between processes [15, 1] can be avoided. We plan to extend this work with additional proof techniques required for a system of concurrent processes sharing multiple resources.

REFERENCES

[1] Apt, K.R., Francez, N. and De Roever W.P., "A Proof System for Communicating Sequential Processes", ACM Transactions on Programming Languages and Systems 2, 3, Jul 1980, 359-385.

[2] Atkinson, R.R. and Hewitt, C.E., "Specification and Proof Techniques for Serializers", IEEE Transactions on Software Engineering SE-5, Jan 1979, 10-23.

[3] Ben-Ari, M., and Pnueli, A., "Temporal Logic Proofs of Concurrent Programs", Technical Report, Tel Aviv University, Nov 1980.

[4] "Reference Manual for the Ada Programming Language", U.S. Department of Defense, July 1980.

[5] Floyd, R.W., "Assigning Meanings to Programs", in Proc. Symposium in Applied MAthematics, Schwartz, J.T. (ed.) 19-32, 1967.

[6] Guttag, V., Horowitz, E., and Musser, D., "Abstract Data Types and Software Validation", Communications of the ACM 21, 1048-1064, Dec 1978.

[7] Hoare, C.A.R., "AN Axiomatic Basis for Computer Programming", Comm. of the ACM, 12, 576-580, 1969.

[8] Hoare, C.A.R. "Monitors: An Operating System Structuring Concept", Comm. of the ACM, 17, 540-557, Oct 1974.

[9] Howard, J.H., "Proving Monitors", Communications of the ACM 19, 549-557, May 1976.

[10] Ichbiah, J.D., et al., "Rationale for the Design of the Ada Programming Language", Sigplan Notices 14, 6, June 1979.

[11] Gouge, V.D., Kahn, G. and Lang, B., "On the Formal Definition of Ada", Rivista di Informatica X n, Mar 1980, 5-14.

[12] Keller, R.M., "Sentinels: A Concept for Multiprocess Coordination", June 1978, UUCS-78-104, University of Utah.

[13] Lamport, L., "'Sometime' is Sometimes 'Not Never'", Proc. Seventh Annual Symposium on POPL, Jan 1980, 174-185.

[14] Manna, Z., and Pnueli, A., "Verification of Concurrent Programs: Temporal Proof Principles", Technical Report, Stanford University, Sep, 1981.

[15] Owicki, S. and Gries, D., "Verifying Properties of Parallel Programs: An Axiomatic Approach", Communications of the ACM 19, May 1976, 279-284.

[16] Owicki, S. and Lamport, L., "Proving Liveness Properties of Concurrent Programs", ACM Transactions on Programming Languages and Systems, 4, 455-495, July 1982.

[17] Pnueli, A., "The Temporal Semantics of Concurrent Programs", in "Semantics of Concurrent Computation", Springer Lecture Notes in Computer Science 70, June 1979, Springer-Verlag, 1-20.

[18] Pnueli, A., "On the Temporal Analysis of Fairness", Proc. Seventh Annual Symposium on POPL, Jan 1980, 163-173.

[19] Ramamritham, K. and Keller, R.M., "Specifying and Proving Properties of Sentinel Processes", Proc. 5th International Conference on Software Engineering, 374-382, March 1981.

[20] Ramamritham, K. "Proof Techniques for Resource Controller Processes", COINS Technical Report, University of Massachusetts, June 1982.

APPENDIX

Proof of Correctness of Mutual Exclusion Translation Rule

The rule is as follows:

```
∀op1,op2,
{ []{service(op1) =>  ~[active(op2) V wait_active(op2)]} &
  []{service(op2) =>  ~[active(op1) V wait_active(op1)]} }
 =>
  [] ~ {active(op1) & active(op2)}
```

Proof: Since all service phases are uninterruptible, for any op1,op2, op1≠op2,

```
[]{service(op1) => ~service(op2) UNTIL after(service_phase|op1)}.
```

However,

```
[]{after(service_phase|op1) => wait_active(op1)}
```

which by the hypothesis and the definition of wait-active,

```
[]after(service_phase|op1) => ~service(op2) UNTIL active(op1)}.
```

Again using the hypothesis,

```
[]{active(op1) => ~service(op2) UNTIL after(active_phase|op1)}
```

Using T9 and the above statements,

```
(10) []{service(op1)
           => ~service(op2) UNTIL after(active_phase|op1)}
```

By the hypothesis,

```
[]{service(op1) => ~active(op2)}
```

which by the definition of the predicate active gives

```
[]{service(op1) => [~active(op2) UNTIL service(op2)}
```

Thus, using (10) and T8,

```
[]{service(op1) => ~active(op2) UNTIL after(active_phase|op1)}
```

Thus by T10 and the above statements,

```
[]{active(op1) => ~active(op2)}
```

Proof of Correctness of the Priority Translation Rule

This rule states that

```
[]{service(op1) => ~req(op2)}
  =>
[]{[req(op1) & req(op2)] => [service(op1) ONLYAFTER service(op2)]}
```

Proof: Assume the hypothesis. By T4

```
[]{req(op2) => [~service(op1) UNTIL ~req(op2)]}
```

By definition of req,

```
[]{req(op2) => req(op2) UNTIL service(op2)}
```

Thus, if

```
req(op1) & req(op2),
```

then by T8 and the above statements,

```
~service(op1) UNTIL service(op2)
```

that is

```
service(op1) ONLYAFTER service(op2).
```

Proof of the Single Slot Buffer Controller

For the sake of brevity, not all steps in the proof are shown. Reference will be made to the appropriate parts of the semantics of resource controller tasks in Ada (see section four). The reader is referred to [20] for the complete proof. Before we start with the proof, using the semantics of the operations in section four, we identify the code corresponding to individual phases of operations.

```
{at(LRS1) & start_read_q'first=r} <=>
{current_operation=r & at(service_phase|r)}

{current_operation=r & after(LRS2)} <=> after(service_phase|r)

at(LRA) <=> ∃r∈Read, at(active_phase|r)

after(LRA) <=> ∃r∈Read, after(active_phase|r)

{at(LRT1) & end_read_q'first=r} <=>
 {current_operation=r & at(term_phase|r)}

{current_operation=r & after(LRT2)} <=> after(term_phase|r)
```

Thus statement LRA corresponds to the active phase of Read operations. Similarly, LDS, LDA and LDT correspond to the service, active and termination phases of Deposit operations respectively. LMS and LMT correspond to the service and termination phases of Remove operations. Remove operations have a null active phase.

The execution context of all phases except the active phase of Read operations is the controller process and hence the execution of these phases cannot be interrupted.

Following are some liveness properties concerning each alternative within the SELECT statement.

```
(11)  {at(LRS1) & start_read'COUNT>0 & status=full}
       =>
      <>{at(LS) & #active_reads>0 & status=full}

      {at(LRT1) & end_read'COUNT>0 & status=full & #active_reads=c}
       =>
      <>{at(LS) & #active_reads=c-1 & status=full}

      {at(LDS) & Deposit'COUNT>0 & status=empty & #active_reads=0}
       =>
      <>{at(LS) & #active_reads=0 & status=full}

      {at(LMS) & Remove'COUNT>0 & status=full & #active_reads=0}
       =>
```

<>{at(LS) & #active_reads=0 & status=empty}

These follow from the semantics of ACCEPT, assignment, DO..END, and LOOP statements. Also, by applying (S4), we have the following implications:

(12) after(LRS1) => (status=full)

after(LDS) => {status=empty & #active_reads=0}

after(LMS) => {status=full & #active_reads=0}

<u>Step-1:</u> <u>Verification of Resource State Changes</u>

Applying standard axiomatic techniques, it is trivial to prove the following:

∀m∈Remove, ∀d∈Deposit,

after(term_phase|d) <=> after(LDT) => (status=full)

after(term_phase|m) <=> after(LMT) => (status=empty)

<u>Step-2:</u> <u>Verification of Invariant Properties</u>

<u>Substep-1</u>: In this substep, we derive the constraints imposed by the resource controller on servicing requests for access. By applying Inference-Rule-I to Deposit, Remove and Read operations, we can derive the following constraints imposed by the controller.

```
(13) ∀r∈Read, ∀d∈Deposit,      service(r) => ~active(d)
     ∀r∈Read, ∀m∈Remove,      service(r) => ~active(m)
     ∀d1,d2∈Deposit,d1≠d2,    service(d1) => ~active(d2)
     ∀d∈Deposit, ∀m∈Remove,   service(d) => ~active(m)
```

The WHEN clauses associated with individual ACCEPT statements are also utilized to derive additional constraints imposed by the controller on the service of operations. From the semantics of operations executed in exclusion, for a Deposit operation d,

service(d) <=> {at(LDS) & Deposit_q'first=d}

By (S4) in the semantics of the SELECT statement,

at(LDS) => {#active_reads=0 & status=empty}

Thus we have

```
     ∀d∈Deposit, service(d)
                   =>
                   Deposit_q'first=d & #active_reads=0 & status=empty
```

Including the conditions derived from (13) above, we get the overall conditions imposed by the SSB controller on servicing deposit operations.

```
(14)  ∀d∈Deposit, service(d) =>
                    {Deposit_q'first=d & #active_reads=0 &
                     status=empty &
                     ∀d1∈Deposit, d≠d1, ~active(d1) &
                     ∀m∈Remove, ~active(m)}
```

Substep-2: In this substep we derive constraints imposed by the specifications of mutual exclusion, priority and others that affect operation service. Applying the translation rules presented in section five,

```
(15)  ∀d∈Deposit,
           service(d) =>
            {req(d) &
             status=empty &
             ∀d1∈Deposit, d≠d1, ~active(d1) & ~wait_active(d1) &
             ∀m∈Remove, ~active(m) & ~wait_active(m) &
             ∀r∈Read, ~active(r) & ~wait_active(r)}
```

Substep-3 This substep involves showing that conditions derived in substep one imply those derived in substep two. At this point we resort to the relationships between predicates on counters and queues and the abstract predicates "req" and "active".

From Q3 (in section four) we can infer that

```
     ∀d∈Deposit, [Deposit_q'first=d] <=> req(d)

     Deposit_q'COUNT=0 => ~∃d∈Deposit, req(d)
```

From C2 (in section four),

```
     #active_reads=0 => ~∃r∈Read, [wait_active(r) V active(r)].
```

Applying the above implications, the constraints imposed by the resource controller on servicing Deposit operations becomes

```
    ∀d∈Deposit,
         service(d) =>
          {req(d) &
           status=empty &
           ∀d1∈Deposit, d≠d1, ~active(d1) & ~wait_active(d1) &
           ∀m∈Remove, ~active(m) & ~wait_active(m) &
           ∀r∈Read, ~active(r) & ~wait_active(r)}
```

which is equivalent to the necessary conditions derived in substep two for Deposit operations (15).

Proceeding in a similar manner, we find that for read operations priority specifications may not hold at all times. After simplifying the constraints derived in substeps one and two, the following remains to be shown:

∀r∈Read, service(r) => [∀m∈Remove, ~req(m) & ∀d∈Deposit, ~req(d)]

This constraint was derived from the priority specifications. Notice that

Deposit'COUNT=0 & Remove'COUNT=0

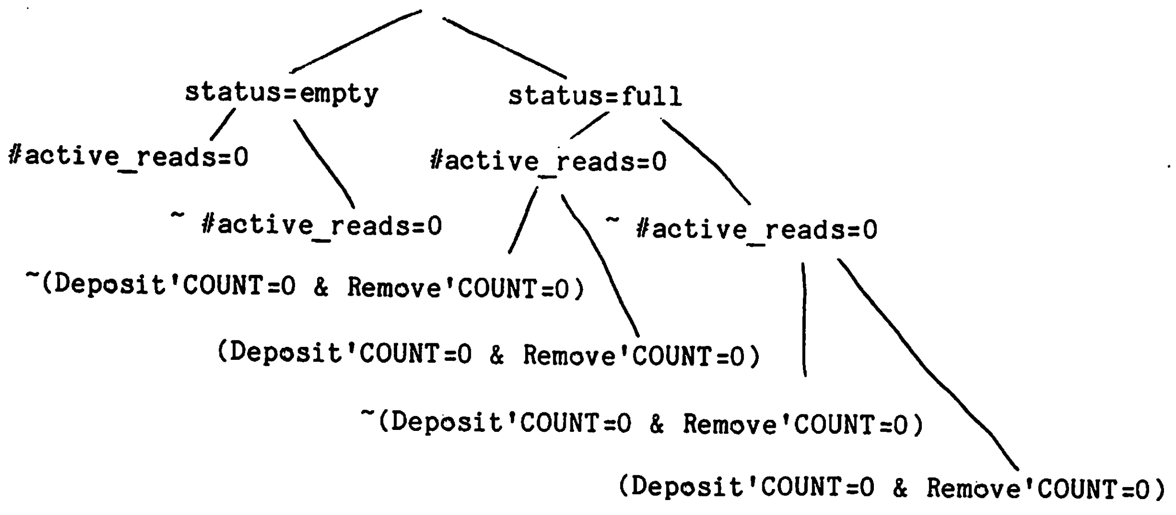
is among the conditions attached to the "ACCEPT Read" statement. However, by (S4) in the semantics of the SELECT statement, this condition may not hold when a Read operation is serviced. Thus, priority specifications may not be satisfied at all times. (One could say that priority specifications may not be satisfied for atmost one execution of the SELECT loop. We do not consider this issue any further.)

Step-3: Verification of Fairness Criteria

The first step in the verification of fairness is to derive the set of disjoint conditions that can hold at the beginning of the SELECT statement. The distinct conditions appearing in the guards are

(status=empty),
(status=full),
(#active_reads=0), and
(Deposit'COUNT=0 & Remove'COUNT=0).

Each branch of the following tree depicts one element of the set of disjoint conditions existing at the beginning of the SELECT statement.


status=empty
status=full
#active_reads=0
~ #active_reads=0
#active_reads=0
~ #active_reads=0
~(Deposit'COUNT=0 & Remove'COUNT=0)
(Deposit'COUNT=0 & Remove'COUNT=0)
~(Deposit'COUNT=0 & Remove'COUNT=0)
(Deposit'COUNT=0 & Remove'COUNT=0)

(Note that by C1, ~#active_reads=0 is equivalent to #active_reads>0.) Applying S2 to each of these conditions, we have six possible combinations of open alternatives and the conditions under which they occur, for example, when (status=empty) and (#active_reads=0), Deposit and end_read alternatives are the only ones open. Let us name the six conditions, E1 to E6. Thus, we have

[]{at(LS) => {E1 V E2 V E3 V E4 V E5 V E6}}

By (11), whichever alternative is chosen, control comes back to LS. Also, one of the guards (namely, for end_read) is always true. Thus by inference rule (S8) and T4,

[]<>{at(LS) & ~OAD & (E1 V E2 V E3 V E4 V E5 V E6)}

Proof of Fairness to Deposit Operations

The fairness required for Deposit operations is

∀d∈Deposit, {req(d) & []<>(status=empty)} => <>service(d)

i.e. if the buffer becomes empty repeatedly, then a Deposit request should be serviced.

We will make use of the following lemma which ensures that if always the first request in a queue is serviced, every request in that queue will be serviced.

∀op, [](op=OPC_q'first) => <>service(op)

=>

∀op, [](op=OPC_q[i]) => <>service(op)

This can be proven by induction on the length of the queue and follows from the semantics of the ACCEPT statement. By the above lemma, it is sufficient to show that the first request in Deposit_q will indeed be serviced.

Assume

req(d) & []<>(status=empty)

where d is deposit_q'first. Suppose d is never serviced, i.e., []~service(d). Since

req(d) => [req(d) UNTIL service(d)]

by T6, []req(d), which by the definition of queues (Q3) results in

[]Deposit'COUNT>0.

Combining the above using T5 we have

(16) []{Deposit'COUNT>0 & ~service(d)}.

Applying steps 3.2 through 3.4 of section five, to each of the six conditions, we prove that request d will be serviced irrespective of the condition that holds at the beginning of the SELECT statement. This is the required contradiction.

Condition-E1: open_alternatives={deposit,end_read}
& status=empty & #active_reads=0

By (16), []deposit'COUNT>0 and so there is a waiting entry call for deposit.

Case-1.1: at(LDS) & status=empty & #active_reads=0 &
Deposit'COUNT>0

Using the definition of operations executed in exclusion,

service(d)

where d is Deposit_q'first. This contradicts (16).

Case-1.2: at(LRT1) & status=empty & #active_reads=0 &
end_read'COUNT>0

which by (11) results in

<>(at(LS) & #active_reads=-1)

contradicting the definition of counters (C1) that

[](#active_reads$\geq$0).

Hence this case cannot arise. (Notice that for end_read'COUNT to be positive, #active_reads should be positive too.)

Condition-E2: open_alternatives={end_read}
& (status=empty) & (#active_reads>0)

It is shown in [20] that (a) if #active_reads>0 then <>end_read'COUNT>0 and (b) by inducing on #active_reads and by using the semantics of the SELECT loop,
<>{at(LS) & #active_reads=0}

which using the tree constructed earlier gives
<>{at(LS) & #active_reads=0)}
=> <>{(at(LS) & #active_reads=0 & (E1 V E5 V E6)}

Consideration of E1 resulted in a contradiction. E5 and E6 will be discussed later.

Condition-E3: after(LRT1) & (status=full) & (#active_reads>0) &
~(Deposit'COUNT=0 & Remove'COUNT=0)

The analysis is similar to condition 2. (Note that the predicates on the counts were not utilized in the analysis of condition 2.)

<u>Condition-E4</u>: after(LRS1) & after(LRT1) &
status=full & #active_reads>0 &
(Deposit'COUNT=0 & Remove'COUNT=0)

By (16), since []deposit'COUNT>0, this condition cannot occur.

<u>Condition-E5</u>: after(LMS) & after(LRT1) &
status=full & #active_reads=0 &
~(Deposit'COUNT=0 & Remove'COUNT=0)

If []Remove'COUNT=0, then []status=full, contradicting the premise of the fairness statement for deposit. Hence eventually, there will be a remove request.

<u>Case-5.1</u>: at(LMS) & status=full & #active_reads=0 &
Remove'COUNT>0

By (11),

<>{at(LS) & status=empty & #active_reads=0}

which will result in condition E1 considered earlier.

<u>Case-5.2</u>: at(LRT1) & status=full &
#active_reads=0 & end_read'COUNT>0

As in Case-1.1, this will lead to a contradiction with []($\#active\_reads \geq 0$).

<u>Condition-E6</u>: after(LMS) & after(LRS1) & after(LRT1) &
(status=full) & (#active_reads=0) &
(Deposit'COUNT=0 & Remove'COUNT=0)

By (16), since []deposit"COUNT>0, this condition cannot occur.

Under each applicable condition, we showed that at least one of the open alternatives will have an entry call. Also, we proved that whichever possible open alternative is chosen from each condition, there is a contradiction with (16) or []($\#active\_reads \geq 0$). Thus our assumption []~service(d) was wrong and hence by T3, <>service(d), i.e., the first request in Deposit_q will be serviced. The proof of the fact that every deposit request will eventually be serviced follows from the lemma.

Proof of fairness for Remove operations is along the same lines as for Deposit operations and is hence omitted.

In general, we would also be interested in showing that every operation will terminate. Statements in (11) prove this for Deposit and Remove operations. The same can be proven for Read operations through an analysis similar to the proof of fairness above.