SEMANTICS OF RESOURCE CONTROL MECHANISMS

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

COINS Technical Report 81-39
December 1981

ABSTRACT

We have introduced a model for expressing the behavior of processes that control shared resource access in 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. The execution of an operation goes through four phases, namely the request, initiation, active and termination phases. A formal framework is provided for delimiting these phases for a given operation, for specifying the execution context of these phases and for stating the temporal ordering of phases of concurrent operations. Usefulness of the model is illustrated by considering resource controller tasks in Ada. Specification and verification of resource controllers is examined in light of this model.

SPECIAL SYMBOLS

∀	The greek symbol ∀. To be read "for all".
∃	The greek symbol ∃. To be read "there exists".
∈	The greek symbol ∈. To be read "is an element of".
~	The symbol for "not".

1. INTRODUCTION

The ever-decreasing cost of hardware has allowed computer users to possess resources dedicated to their individual needs. It is no longer necessary for users to share a single expensive resource. Nevertheless, the nature of certain types of computing such as those found in airline reservation systems and automated offices makes it necessary for resources such as files and queues to be shared among multiple users.

Thus there is a need for protection and synchronization mechanisms to ensure the correct functioning of shared resources. To provide these mechanisms, an answer to the question, "Who can access the resource, when and how?" is essential. Protection mechanisms are responsible for the who and how of shared resource access whereas synchronization mechanisms are responsible for when resource accesses take place. Here we confine our attention to synchronization issues.

There are essentially two methods to ensure proper resource sharing:

1. All processes are jointly responsible for the consistency of the shared resources, as in [Reed79]. Typically, a process interacts with the others and accesses a shared resource only when it is safe to do so.

2. For every shared resource, a specific program module is made responsible for managing that resource. Individual processes interact only through these modules, as in [BrinchHansen78]. While limiting the interactions, this model contributes to the modularity of the system.

This paper is devoted to synchronization using the latter method.

A resource can be considered to be an abstract data type consisting of the resource definition and the access operations on the resource [Guttag77]. 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 [Hoare74]. Serializers [Atkinson79], sentinels [Keller78] and the Ada tasking facility employing the rendezvous mechanism [DoD80] evolved from the monitor mechanism. In this paper, we refer to the above mechanisms as resource controllers. The monitor is a passive module in that user processes themselves execute the access operations. In the later mechanisms, an active process is associated with the shared resource and it is this process that executes the operations on the caller's behalf. Many of the differences between the mechanisms arise due to the restrictions imposed by them on the access operations, over and above those mandated by the resource. For example, in monitors all access operations are executed in mutual exclusion even though this is not always necessary. Some of these restrictions have been introduced to simplify the mechanisms while others are necessitated by the underlying execution model. In light of the differences, the relative merits of the proposed mechanisms is not always obvious due to the absence of an appropriate model for resource sharing. Here we attempt to remedy this situation by proposing a formal model for shared resources. It facilitates a comparison of the various resource control mechanisms by abstracting out details not pertinent to resource control.

In the model that we propose, an access operation goes through a number of phases. They are the request phase, the initiation phase, the active phase and the termination phase. Differences between resource control mechanisms appear in terms of (1) the temporal ordering of the phases of operations when users make concurrent requests, and (2) the execution context of the phases, i.e., whether the process which makes

the request, the resource controller, or a third process executes a phase. We show that for a particular mechanism, resolution of these two issues leads to its precise definition. We have also explored specification and verification of resource controllers in light of the proposed model.

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 [Pnueli79], on the other hand, lend themselves to specifying individual properties of interest without specifying the complete behavior. Due to the ability of temporal logic to deal with invariant behavior (through the "always" operator), with eventual behavior (through the "eventually" operator), and with ordering relationships (through the "until" operator), we find temporal logic to be a natural formalism to deal with resource control mechanisms.

The proposed model for shared resource access is explicated in Section two. The model is first informally defined and then formalized using temporal logic as the tool. In Section three, Ada's rendezvous mechanism is examined in the context of our model. This consideration leads us to the formal definition of the behavior of resource controller tasks in Ada. In Sections four and five, specification and verification of resource controllers is explored. In particular, information that needs to be specified and a method for the verification of correctness of extant resource controllers with respect to these specifications are examined. In the final Section, previous approaches to the issues under discussion here along with the contributions of this approach are outlined.

2. A MODEL FOR RESOURCE SHARING IN CONCURRENT SYSTEMS

Informal definition of the model

A shared resource can be considered to be an abstract data type [Guttag78] comprised 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. Operations on a resource can be executed only if the controller for that resource permits their execution. Execution of an operation goes through four distinct phases. They are:

1. Request phase,
2. Initiation phase,
3. Active phase, and
4. Termination phase.

They occur according to the above sequence. An informal definition of the phases is presented now. A precise definition follows in the next section.

Concurrent processes access a shared resource by requesting the execution of one of the access operations. *When* the controller for that resource will permit execution depends on the state of the shared resource, priority associated with the request, and other criteria.

These determine the necessary conditions for executing an operation.

The request phase for an operation begins when a resource controller recognizes that a user needs to execute that operation. The request phase ends when the controller's internal data structures reflect the fact that a request is waiting for service.

The initiation phase begins when and if necessary conditions hold and the resource controller decides to permit the execution of the operation. At the end of the initiation 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 "initiation" is equivalent to "granting of permission".

The active phase begins when the initiation 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 when 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.

In order to exemplify this model and for use in further discussion, we use a standard resource control problem, viz., Readers and Writers [Courtois71]. We use the following version of the problem: "Read" and "Write" are the two classes of access operations defined on a shared data structure. Read operations do not modify the state of the data while write operations normally do. Write requests have priority over read requests.

Now we present the skeletal code for a controller of the data. The variable #active_reads (#active_writes) is initially 0 and keeps count of active read (write) operations. Read_queue (write_queue) is a queue for waiting read (write) requests and is initially empty.

Definition of operation READ:

```
     -- the controller recognizes a user's need to read
RRP: ENQUEUE "req" INTO read_queue;

     WHEN (EMPTY(write_queue) AND #active_writes=0) DO

RIP: BEGIN
      DEQUEUE "req" FROM read_queue;
      #active_reads := #active_reads + 1;
     END;

RAP: --  perform read access

RTP: #active_reads := #active_reads - 1;
```

Definition of operation WRITE:

```
     -- the controller recognizes a user's need to write
WRP: ENQUEUE "req" INTO write_queue;

     WHEN (#active_reads=0 AND #active_writes=0) DO

WIP:  BEGIN
        DEQUEUE "req" FROM write_queue;
        #active_writes := #active_writes + 1;
      END;

WAP: --  perform write access

WTP: #active_writes := #active_writes - 1;
```

Phases of an access operation can be associated with statements in the code for the controller of the operation. For example, in the code above, RRP (WRP) stands for Read (Write) Request Phase, RIP (WIP) stands for the Read (Write) Initiation Phase, RAP (WAP) stands for the Read (Write) Active Phase, and RTP (WTP) stands for the Read (Write) Termination Phase. The conditions following the WHEN clause in the code for read and write are the necessary conditions for read and write

operations respectively.

For the moment let us not be concerned with necessary details such as

- which process executes each of the phases,
- control of access to the variables used for resource control,
- which of the waiting reads and writes are initiated, etc.

We will subsequently attend to these details. Here it suffices to note that a typical access operation goes through the four phases mentioned earlier.

Formal definition of the model

In order to formally define the model, we introduce some notation.

p¦op

will be used to refer to phase p of operation op. Thus,

active_phase¦a

refers to the active phase of operation a. Earlier we noted that the phases that occur during the execution of an operation in some class can be associated with statements in the resource controller code for that class. To reason about the execution of these statements, the following predicates will be utilized.

Predicates associated with executable statements:

Given a statement S that is executed by some process,

at(S) IFF control of that process is at the beginning of S.

in(S) IFF control is within S.

after(S) IFF control is immediately following S. Given a statement sequence S;T, after(S) <=> at(T)

These three predicates are mutually exclusive and they become true in

the above order. We do not make any assumptions about statement S. For instance, S could be a composite of statements such as a BEGIN..END construct. The formal definition of the language construct corresponding to S would specify how the component statements are affected by the execution of S. However, the following can be said regarding the flow of control through any executable statement S. Assuming that the underlying scheduler of processes is fair, if control is at the beginning of a statement then control will eventually be within the statement, and if the statement is known to terminate, then control will eventually reach the end of the statement.

Now we can state

at(active_phase|a)

to assert that control is at the beginning of a's active phase. To specify the relationship between phases, we need a formalism through which it is possible to state the behavior of a resource controller throughout its execution. We find temporal logic to be appropriate for this purpose.

Temporal logic:

Temporal logic was introduced by Pnueli to reason about the invariant and time-dependent properties of concurrent programs [Pnueli79]. Concurrency is modeled by a nondeterministic interleaving of computations of individual processes. Each computation changes the system state consisting of values assigned to program variables and the instruction pointer of each of the processes. Using temporal logic operators, we can specify and reason about the properties of the sequence of states that results from the execution of the concurrent processes. The truth value of P at some point j in the execution

sequence is denoted by Pj. We employ three primitive temporal operators. Of them [] and <> are unary operators whereas UNTIL is a binary operator. Since temporal logic is an extension of propositional calculus, a temporal logic statement can involve the usual logical operators V (or), & (and), ~ (not) and => (implication) besides the temporal operators.

The operator [] is pronounced "always". []P states that P is true now and will remain true throughout the future.

([]P)i IFF ¥j≥i, Pj.

where current state is the ith element in the state sequence.

The operator <> is pronounced "eventually" and is the dual of P in that

(<>P)i IFF ∃j≥i, Pj.

Thus, <>P if P is true now or will be true sometime in the future.

The operator UNTIL has the following interpretation:

(P UNTIL Q)i IFF ¥j≥i, ([¥k i≤k≤j, ~Qk] => Pj).

Thus, as long as Q is false, P will be true. In other words, P remains true until Q becomes true.

An invariant property I can now be specified as []I. A requirement such as "every request will be serviced" can be specified as

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

The UNTIL operator is typically used for expressing temporal orderings. For example, the fact that a service can not be done 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) IFF (~P UNTIL Q).

Temporal ordering of phases in an operation:

The four phases associated with an operation "a" are totally ordered in time as follows:

at(init_phase|a) ONLYAFTER after(req_phase|a)

at(active_phase|a) ONLYAFTER after(init_phase|a)

at(term_phase|a) ONLYAFTER after(active_phase|a)

The first statement reflects the possibility of a request having to wait before initiation. In the last two statements, we have used ONLYAFTER instead of IFF since the active and termination phases could be executed by a process other than the controller. 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 a.

Predicates associated with the phases of an operation:

When a request is present for operation a, the predicate req(a) is true. The behavior of this predicate is expressed through the following axioms.

req(a) ONLYAFTER at(req_phase|a)

after(req_phase|a) => {req(a) UNTIL [at(init_phase|a) & req(a)]}

after(init_phase|a) => []~req(a)

Through these axioms we have stated the following:

A request for an operation can be said to be present only after

the request phase for that operation begins.

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

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

Thus it is not possible to determine the truth of req(a) when control is within the request phase or the initiation phase. This is intentional and is necessitated by the manner in which resource controllers handle requests.

The predicate init(a) is true if and only if control is at the beginning of the initiation phase for a.

init(a) <=> at(init_phase¦a)

Also, an initiation phase should not begin unless the appropriate request is present. Therefore the following should hold at all times.

init(a) => req(a)

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

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

term(a) <=> after(term_phase¦a)

To summarize,

req(a) is true when operation a is waiting to be initiated.

init(a) is true when permission is granted for executing a.

active(a) is true when the access defined by a takes place.

term(a) is true when termination phase is completed.

The reader would have noticed that the model does not specify the

process which executes each phase. If the controller is an active process, as in Ada, then not all phases need be executed by the resource controller itself. By itself not executing a phase, the resource controller becomes available for the execution of phases of other operations. In particular, by itself not executing the active phase of operations, the controller makes concurrent accesses possible. In the context of resource sharing, the process that executes a phase could be the process requesting access to the resource, the resource controller, or a third process activated by the controller. In addition, the model does not specify the overall ordering of the phases of operations when users make requests concurrently. These two factors are functions of the nature of the shared resource, the access operations defined on the resource and the necessary conditions for their execution. In practice, however, the mechanism chosen for implementing resource control imposes further restrictions on the execution of the phases. For instance, in monitors [Hoare74], all accesses take place in mutual exclusion.

Our purpose behind modeling shared resource access is twofold:

1. Provide an abstract view of shared resource control.
2. Utilize the model to specify and verify the behavior of resource control mechanisms.

In order to achieve (2) for a particular mechanism, one has to delimit the phases of access operations performed by the resource control mechanism as well as define the relationship between the phases of different operations. Delimiting the phases basically involves identifying when at(p|a) and after(p|a) hold for every phase p of every access operation a. That in turn will define when the predicates req, init, active and term will be 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 exexute 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 = {request_phase, initiation_phase,
active_phase, termination_phase}
SPR = {calling_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.

Thus if the active phase of write operations is executed by the controller of reads and writes then

execution_context(write,active_phase)=controller_process

Ordering relationships between phases:

We say that a phase p executed by a controller cannot be interrupted if from the beginning to the end of execution of phase p, control cannot reach the beginning of any other phase executed by the same controller.

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

Extending this to a sequence of phases, we say that a sequence of phases (p1,p2,..,pn) cannot be interrupted if from the beginning of execution of phase p1 to the end of execution of pn, control cannot reach the beginning of any phase (executed by the same controller) not in the sequence. Formally,

(p1,p2,..,pn) cannot be interrupted

IFF

$\forall j\ 1 \leq j \leq n, \forall p \neq pj,$

at(p1) => [at(p) ONLYAFTER after(pn)]

It should be obvious to the reader that any phase executed by the resource controller process should not be interrupted. Recall that our definition of interruption relates only to phases executed by a resource controller. Thus the fact that a phase p cannot be interrupted does not preclude the interruption of p to make the processor available for another process, in which case, execution of the resource controller process will subsequently continue from the point where such *an* interruption took place.

The main function of a resource controller is to permit accesses when the necessary conditions are true. In other words, the resource controller is responsible for the initiation of access operations and hence initiation phases of operations are executed by the resource controller process itself. Thus we have,

∀OPC, execution_context(OPC,initiation_phase)=controller_process.

Initiation_phase cannot be interrupted.

Our model presents a framework utilizing which the behavior of resource control mechanisms can be defined. In particular, it provides means for delimiting the phases that constitute the operations, for specifying the execution context of the phases and stating the ordering relationships between phases. As an example of its application we will now look at resource controller tasks in Ada.

3. RESOURCE CONTROLLER TASKS IN ADA

In Ada, tasks are the program units for concurrent programming. An entry definition within a task can be thought of as an operation on the resource controlled by the task. 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. (For details of Ada's tasking facility see [DoD80].)

As in monitors, Ada has a built in mutual exclusion mechanism for performing a rendezvous. Thus concurrent executions of access operations, such as read in the readers and writers problem, have to be engineered by programming the operations as procedures with appropriate entry calls before and after the code for the operation. Unlike monitors, resource controllers in Ada are active entities in that the resource controller process itself performs the accesses. Skeletal code for the read-write controller appears on the following page.

The resource being accessed, the operations on it and the controller are defined within a package. A call on the procedure Write translates into a call on the entry corresponding to Write whereas 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.) Henceforth we will say that "a write operation is executed through an entry call" and that "a read operation is executed through a procedure call".

```
PACKAGE rw IS
  [specification of resource]
  PROCEDURE read [arguments];
  PROCEDURE write [arguments];
END rw;

PACKAGE BODY rw IS

TASK rw_controller IS
 ENTRY start-read;
 ENTRY end-read;
 ENTRY write [arguments];
END rw_controller;

TASK BODY rw_controller IS
 #active_reads, #active_writes : INTEGER := 0;

 BEGIN
  LOOP
   SELECT
       WHEN write'count=0 =>
        ACCEPT start-read;
        DO
         #active_reads := #active_reads + 1
        END
    OR
        ACCEPT end-read;
 DO
         #active_reads := #active_reads - 1;
        END
    OR
       WHEN #active_reads=0 =>
        ACCEPT write DO
         -- perform write operation
        END;
   END SELECT;
  END LOOP;
 END rw_controller;

 PROCEDURE read [arguments] IS
 BEGIN
  rw_controller.start-read;
  -- perform read operation
  rw_controller.end-read;
 END;

 PROCEDURE write [arguments] IS
 BEGIN
  rw_controller.write [arguments];
 END;

 END rw;
```

Recall our use of #active_writes to keep count of the number of active write operations. From the above code we note that due to the mutual exclusion of the rendezvous mechanism,

[] (0≤#active_writes≤1)

and that except when a rendezvous for a write operation is taking place,

#active_writes = 0.

Hence the necessary conditions for read and write have been simplified as shown. Also, changes to #active_writes have been eliminated.

For each entry within a task, a distinct queue is used for entry calls that are waiting to be accepted. Thus the elements in the queues correspond to the processes that are waiting for rendezvous to take place. When an entry call (in other words, a user's expression of need for access) is recognized, it is placed in the appropriate queue. Thus 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 attribute "count" of an entry can be used to determine the number of waiting entry calls. Thus

write'COUNT

keeps count of the number of waiting write requests.

Semantics of Resource Controller Tasks in Ada

Providing precise semantics for resource controller tasks involves delimiting the phases of the operations performed by a task as well as defining the relationships among the phases of different operations. Towards this end we distinguish between the following:

1. An access operation executed through an entry call; this

permits the mutual exclusion of the operation with the rest of the operations.

2. An access operation executed through a procedure call; this permits concurrent execution of operations.

To be concrete, let us assume a class of operations C. Cq is the queue for waiting calls on 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;
SA: <perform operation C>
    stop-C;
    END;
```

Within the task body, the code for entries start-C and stop-C will be defined as follows:

```
    WHEN <necessary conditions for C> DO
     ACCEPT start-C
      DO
SI:  <modify internal variables to reflect initiation>
      END;

    ACCEPT stop-C
     DO
ST:  <modify internal variables to reflect termination>
     END;
```

When control in an Ada task reaches the statement "ACCEPT start-C", it remains there until an entry call on start-C occurs. (Whether control will eventually be within the ACCEPT statment, once the call occurs, depends on whether the statement is within a SELECT clause or not. This issue is considered later.) Once control is within the ACCEPT statement, the first element in the corresponding queue is removed and the statements following the ACCEPT statement are executed. Based on this, the behavior of the resource controller task can be given as follows (statement labels in the code above are utilized here).

Let S stand for the statement

ACCEPT start-C

and Cq[i] indicate the i-th element in the internal queue for entry start-C. Then,

```
at(S) => [at(S) UNTIL C'count>0]

{at(S) & Cq[1]=c}
 =>
{req(c) &
 <>in(S) => <>init(c) &
 after(S) ONLYAFTER init(c) &
 []{after(S) => ~req(c)}}

after(SI) <=> after(init_phase|c)

at(SA) <=> at(active_phase|c)

after(SA) <=> after(active_phase|c)

at(ST) <=> at(term_phase|c)

after(ST) <=> 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,request_phase)=controller_process
execution_context(C,init_phase)=controller_process
execution_context(C,active_phase)=calling_process
execution_context(C,term_phase)=controller_process
```

The following statements define the constraints on the executions of these phases.

Request phase cannot be interrupted.
Initiation phase cannot be interrupted.
Termination phase cannot be interrupted.

The first statement is necessary since it is the resource controller task that enqueues the requests. The last two statements become necessary given that the initiation and termination phases are executed by the resource controller process. The orderings of phases imposed by

an Ada resource controller task conforms with the above constraints.

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:

```
    WHEN <necessary conditions for C> DO
     ACCEPT C
     DO
SI:  <modify internal variables to reflect initiation>
SA:  <perform operation>;
     END;
ST:  <modify internal variables to reflect termination>
    END;
```

Here again, let S stand for the statement

```
ACCEPT start-C
```

and Cq[i] indicate the i-th element in the internal queue for C. Then,

```
at(S) => [at(S) UNTIL C'count>0]

{at(S) & Cq[1]=c}
 =>
{req(c) &
 after(S) ONLYAFTER init(c) &
 <>in(S) => <>init(c) &
 []{after(S) => ~req(c)}}

after(SI) <=> after(init_phase|c)
          <=> at(SA) <=> at(active_phase|c)

after(SA) <=> after(active_phase|c)
          <=> at(ST) <=> at(term_phase|c)

after(ST) <=> after(term_phase|c)
```

These define the phases of an operation executed as an entry. The execution context of these phases is specified by the following statements.

```
execution_context(C,request_phase)=controller_process
execution_context(C,init_phase)=controller_process
execution_context(C,active_phase)=controller_process
execution_context(C,term_phase)=controller_process
```

The following statements define the constraints on the executions of

these phases.

Request phase cannot be interrupted.
The sequence {initiation phase,active phase}
cannot be interrupted.
Termination phase cannot be interrupted.

Thus, this essentially differs from the first case in the uninterruptibility of the active phase which in turn leads to the mutual exclusion of execution of operations.

In the case of Ada, a process accessing a resource through an entry call waits until the rendezvous is complete, after which it proceeds with its execution. Hence the calling task can execute concurrently with the termination phase of the called operation. Thus, if R is the satement calling operation a,

after(R) ONLYAFTER after(active_phase)

[]{after(active_phase|a) => after(R)}

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

after(R) ONLYAFTER after(term_phase|a)

[]{after(term_phase|a) => after(R)}

Here the calling task can continue execution only after the termination phase is completed.

Due to the sequential nature of the resource controller, the initiation phases of all operations is totally ordered. An important aspect of resource controller behavior concerns the choice made in initiating one request from among those that have satisfied their necessary conditions. In Ada, this manifests itself in terms of the choice made among open SELECT alternatives which have a waiting access request. According to [DoD80] this choice is arbitrary, which in effect

implies that nothing can be said regarding the response of a resource controller task to access requests. We believe that this lack of precise definition of the SELECT statement makes it impossible to verify properties such as termination of access operations, and hence, the progress of tasks. However, in [Ichbiah79], two possible implementations of the SELECT statement are described. Of them, the "order of arrival method" is amenable to a semantic definition. According to this method,

> Among the waiting entry calls, that one whose guard is true and which was the earliest to arrive, is selected.

This satisfies the following property.

$\forall c, []<>\text{necessary-condition}(c) \Rightarrow <>\text{init}(c)$

i.e., if an operation's necessary conditions (for initiation) become true repeatedly, then it will be eventually initiated. This is formally proved in [Ramamritham81b]. Intuitively, since the oldest waiting request is initiated when its necessary conditions are true, if not already initiated, a waiting operation will eventually become the oldest waiting request. Due to its necessary conditions becoming true infinitely often, i.e., repeatedly, the oldest waiting request will eventually have its necessary conditions satisfied when it will be initiated. (The assumption underlying this reasoning is that the truth value of necessary conditions does not change except due to some activity within the resource controller task. This may not always be true if necessary conditions involve global variables, in which case, even such a fairness is not satisfiable.) Thus, if Cg is the guard for an ACCEPT statement S involving entry C, then by the order of arrival method,

$[]<>Cg \Rightarrow <>in(S)$

Note that if S is not within a SELECT statement, no such restrictions apply. In that case if the scheduler of tasks is fair, then

{at(S) & C'count>0} => <>in(S)

To summarize our discussion of resource control tasks in Ada, accesses to a resource encapsulated in a package is through procedure calls or entry calls, the former being appropriate for concurrent accesses and the latter for mutually exclusive accesses. In both cases, we specified when each phase begins and ends, which process executes a phase and the constraints on the execution of phases. The last piece of information in turn determines the allowable orderings of executions of the phases. It was pointed out that the definition (or more correctly, the lack of it) of Ada's SELECT statement made the analysis of "liveness" properties impossible.

4. SPECIFICATION OF RESOURCE CONTROLLER PROBELEMS

We mentioned earlier that most resource controller mechanisms restrict the execution of different phases over and above that required by the problem. How does one determine what the restrictions imposed by a problem are? To answer this question, we need to be able to deal with the specifications for a resource control problem. Our model of resource control gives us a handle on the issue of specifications. Note that out of the four phases, only the initiation phase is constrained to begin only after requisite conditions are met. After initiation, active and termination phases follow unhindered. In order to derive the constraints, specifications for a resource control problem should provide answers to the following questions.

1) How do the request, initiation, active, and termination phases of an operation modify the state of the system?
2) What are the invariant constraints on the resource?
3) What are the necessary conditions for initiating operations?
4) In what order are requests to be initiated?

Answer to (1) can take the form: If phase p of operation a is executed in exclusion, then for a variable v,

```
{at(p|a) & v=initial¥value}
=>
[]{after(p|a) => v=final¥value}
```

Based on the answer to (1), those phases that change the state of a variable should be executed in exclusion. Sometimes it may be necessary to explicitly state which phases are to exclude one another, as in the case of active phases of write operations.

As regards (2), if a predicate p is an invariant of a resource, then it can be specified as []p.

Answer to (3) can be expressed as

```
[]{init(op) => condition}
```

Thus when the operation "op" is initiated, "condition" is required to be true. Some of the conditions on initiating operations arise from considerations such as invariant constraints on the resource, mutual exclusion, priority, etc. As we show in the next section, such implied conditions are derivable from the specifications. Thus the answer to (3) need only involve explicit conditions that need hold when an operation is initiated.

Answer to (4) can take the following forms:
- specification of permitted sequences of operations, for instance through path expressions [Habermann75].
- specification of priority for access operations.
- specification of fairness requirements [Pnueli80].

These can be utilized to order initiations.

In [Ramamritham81a], a high-level language is presented in which answers to the above questions can be expressed. Using that language, the specification for the Readers and Writers problem can be given as follows. Formal meanings of these statements appear in the next section.

```
EXCLUSION
  Write EXCLUDES Read
  Write EXCLUDES Write

PRIORITY
  Write > Read

FAIRNESS
  [] {necessary-condition(w) => <>init(w)}
  []necessary-condition(r) => <>init(r)
```

where r is a read operation and w is a write operation. The fairness required is that a write request that has satisfied its necessary conditions should be eventually initiated. On the other hand, a read request need be initiated only if its necessary condition always remains true. This is due to the priority given to write requests.

5. VERIFICATION OF RESOURCE CONTROLLER TASKS

We saw earlier that a resource controller decides when an access operation should be initiated and that once an operation has been initiated, the active and termination phases follow. Thus any control over resource access should be exercised at the time of initiation. In other words, prior to initiation it should be ascertained that the execution of an access operation will conform to the specifications. Verification involves performing the following steps:

1) Once the necessary conditions for each access operation have been determined from relevant specifications, and the conditions imposed by the resource controller are determined from the code, the following statement should be verified.

   (necessary conditions imposed by the resource controller)
   =>
   (necessary conditions determined from the specifications)

2) Verify that the fairness specified is adhered to.

Here we shall only informally show how steps (1) and (2) can be carried out. First we indicate how the high-level specifications for the Readers and Writers problem can be transformed into the necessary conditions for the read and write operations.

The semantics of the statement

Read EXCLUDE Write

is

[] ~ {active(r) & active(w)}

for all read operations r and write operations w. To translate this into necessary conditions for read and write operations, we now introduce a theorem (proved in [Ramamritham81b]).

```
∀op1∈OPC1, []{init(op1) => ~∃op2∈OPC2, active(op2)} &
∀op2∈OPC2, []{init(op2) => ~∃op1∈OPC1, active(op1)}

=>
```

```
¥op1∈OPC1, ¥op2∈OPC2,
[] ~ {active(op1) & active(op2)}
```

Typically, counters are employed to keep count of the number of active operations. When an operation is active, the counter of the corresponding class of operations is positive. Thus,

```
∃r∈Read, active(r) => #active_reads>0
```

Note that the converse need not be true. From the above statement,

```
#active_reads=0 => ~∃r ∈ Read, active(r)
```

and hence, when a counter is zero, we infer that there are no active operations in the corresponding class. Thus, based on this theorem, the mutual exclusion of reads and writes translates to the following necessary condition.

```
¥r∈Read, [](init(r) => #active_writes=0}
¥w∈Write,[](init(w) => #active_reads=0}
```

The semantics of the priority specification transforms writer's priority over readers into

```
¥r∈Read,  ¥w∈Write,
 [](req(r) & req(w)
    =>
    init(r) ONLYAFTER init(w)}
```

The following theorem (also proved in [Ramamritham81b]) aids in the translation of priority specifications into necessary conditions.

```
¥op1∈OPC1, ¥op2∈OPC2, [](init(op1) => ~req(op2)}

  =>

¥op1∈OPC1, ¥op2∈OPC2, [](req(op1) & req(op2)
                              =>
                               init(op1) ONLYAFTER init(op2)}
```

Typically, resource controllers employ queues for waiting requests. Hence,

```
∃op∈OPC, req(op) => OPC'count>0
```

Thus the priority specification results in the following necessary

condition for a read operation r.

$\forall r \in Read,\ []\{init(r) \Rightarrow write'count=0\}$

The overall necessary conditions for the readers writers problem is obtained by conjuncting the constraints derived from the specifications. It is given by

$\forall r \in Read,\ []\{init(r) \Rightarrow (\#active\_writes=0 \ \& \ Write'count=0)\}$

$\forall w \in Write,\ []\{init(w) \Rightarrow (\#active\_writes=0 \ \& \ \#active\_reads=0)\}$

The fact that these necessary conditions are satisfed by the resource controller task "rw_controller" can be observed by examination of the code. It can be formally proved by deriving the conditions that hold when control is within an ACCEPT statement.

We now briefly look at the verification of fairness. The specification requires that once a write operation has satisfied its necessary conditions, it should be eventually initiated. From the code for the task "rw" we note that once control is inside the loop, if a write request is present and the guard for "ACCEPT write" is true, then the guard for read cannot be true. Thus the ACCEPT statement for write will be executed. The fairness specification for read requires that if the necessary conditions for a read operation are always true then it should be initiated. This is found to be the case since the guard for read and the guard for write cannot be true simultaneously whereby if a read request's necessary conditions are always true, the read request has to be eventually executed.

In this section, we have indicated how our model of resource control aids in the proof of correctness of extant resource control code. In a future paper we will examine verification in greater detail.

6. CONCLUSION AND RELATED WORK

We have 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 the access to proceed, a controller initiates an access operation.
- The code for an access operation is executed only after the controller has initiated 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, one for each phase, 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 [Hoare74], Ada Tasks [DoD80], serializers [Atkinson79] and sentinels [Keller78]. In this paper, we dwelt on 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 state these restrictions completely.

The model adopted in [Laventhal78] associates three events, namely, request, enter and exit, with each access to a resource. The set of events associated with accesses to a particular shared object are assumed to be totally ordered. A description of a controller using this model will not be complete since it does not specify the nature of state transitions that accompany these events. We find it more natural to

view each access to be made up of four phases, as discussed in this paper. The beginning of the request phase, the beginning of the initiation phase, and the end of the termination phase correspond to Laventhal's request, enter and exit events. In our model, no assumptions are made concerning the ordering of the phases associated with concurrent accesses on a data object; however, using the formalism described herein it is possible to specify the orderings imposed by a resource control mechanism.

Of the four phases that occur during the execution of an operation, a resource controller has explicit control over only the initiation phase. The specification for a resource control problem should state how such control should be exercised. In this context, we viewed the problem of specification of resource control and categorized the high-level properties that are consequential for resource control. We briefly looked at the issue of verification, also in light of our model.

This work will be extended in the following directions:

- Here we have focussed only on the primary features in Ada for tasking. Others such as task priorities, delay statements, etc, need be brought under consideration.

- On the surface it appears as though the model is applicable only to modular synchronization mechanisms. However, we believe that any resource access conceptually involves the four phases introduced here. Thus resource sharing through the use of low-level primitives such as semaphores will also be examined.

- Definition of a model for resource sharing is only the first step to handle various issues connected with resource sharing. Two of these, specification and verification were briefly discussed here. These and other issues will be pursued further.

REFERENCES

[Atkinson79] Atkinson, R.R. and Hewitt, C.E., "Specification and Proof Techniques for Serializers", IEEE Transactions on Software Engineering SE-5, Jan 1979, 10-23.

[BrinchHansen78] Brinch Hansen, P., "Distributed Processes: A Concurrent Programming Concept", Communications of the ACM 11, Nov 1978, 934-941.

[Courtois71] Courtois, P.J. Heymans, F. and Parnas, D.L., "Concurrent Control with 'Readers' and 'Writers'", Communications of the ACM 14, Oct 1971, 667-668.

[DoD80] "Reference Manual for the Ada Programming Language", U.S. Department of Defense, July 1980.

[Guttag78] Guttag, V., Horowitz, E., and Musser, D., "Abstract Data Types and Software Validation", Communications of the ACM 21, 1048-1064, Dec 1978.

[Habermann75] Habermann, A.N., "Path Expressions', June 1975, Carnegie-Mellon University.

[Hoare74] Hoare, C.A.R. "Monitors: An Operating System Structuring Concept", Comm. of the ACM, 17, 540-557, Oct 1974.

[Ichbiah79] Ichbiah, J.D., et al., "Rationale for the Design of the Ada Programming Language", Sigplan Notices 14, 6, June 1979.

[Keller78] Keller, R.M., "Sentinels: A Concept for Multiprocess Coordination", June 1978, UUCS-78-104, University of Utah.

[Lamport80] Lamport, L., "'Sometime' is Sometimes 'Not Never'", Proc. Seventh Annual Symposium on POPL, Jan 1980, 174-185.

[Laventhal78] Laventhal, M.S., "Synthesis of synchronization code for data abstractions", S.M. Thesis, M.I.T., June 1978.

[Pnueli79] 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.

[Pnueli80] Pnueli, A., "On the Temporal Analysis of Fairness", Proc. Seventh Annual Symposium on POPL, Jan 1980, 163-173.

[Ramamritham81a] Ramamritham, K. and Keller, R.M., "On Synchronization and its Specification", Springer Lecture Notes in Computer Science 111, June 1981.

[Ramamritham81b] Ramamritham. K., "Specification and Synthesis of Synchronizers ", Ph.D. Thesis, The University of Utah, Aug 1981.

[Reed79] Reed, D.P. and Kanodia, K. "Synchronization with Eventcounts and Sequencers", Communications of the ACM 22, 2, Feb 1978, 115-123.