Efficient Implementation of Ada Tasks

January, 1980

A. N. Habermann
Carnegie Mellon University

and

Isaac R. Nassi
Digital Equipment Corporation

ABSTRACT: A mechanism is described for implementing ACCEPT bodies in Ada as critical sections controlled by P and V-like operations on state variables which are similar to semaphores. It is shown that the implementation makes task switches at the point of rendezvous unnecessary in most cases. The mechanism is general, but allows significant optimizations for straightforward programs.

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pa. 15213 USA

Digital Equipment Corporation
146 Main Street
Maynard Ma. 01754

This research was sponsored in part by the Defense Advanced Research Projects Agency (DOD), ARPA order no. 3597, monitored by the Air Force Avionics Laboratory under contract F33615-78-C-1551.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. government.

1.0 INTRODUCTION

Multitasking in Ada consists of a set of interdependent language constructions built around the concept of a rendezvous. An entry in Ada belongs to a task activation. An accept statement names an entry, and associates the entry name with a body of code. Additionally, there is a list of formal parameters associated with the entry. Intertask communication is accomplished by "calling" an entry with the same notation used to call a procedure. The task activation executing an accept statement and the task calling the entry meet in a rendezvous and stay locked in time until the body of code associated with the accept statement completes its execution, at which point both the calling task and the called task continue asynchronously.

Thus, two scenarios are possible. The calling task can issue the call first, or the task owning the entry can execute an accept for that entry first. A naive implementation might work as follows:

1. Call precedes accept

   A task calls the entry in the other task and blocks. When the called task executes an accept for this entry, it associates the actual parameters with the formals of the entry and executes the body textually associated with this accept. When the body completes, the calling task is made ready to execute and the scheduler is entered to select a task to execute next. The called task continues to be available for scheduling.

2. Accept precedes call

   The task owning the entry executes an accept for the entry and blocks. A task issues an entry call and blocks, after making the first task eligible for scheduling. When this task starts executing, it executes the body, and continues as in case 1.

Observe that either two or three scheduling points are required, depending on which of the above cases applies. At each scheduling point, there is the potential of a task context switch, in which the state of the task must be saved, and the context of another task restored and made ready to run. Fortunately, there is a better way. Our solution attempts to reduce the number of context swithces by allowing the accept body to run as part of the caller's thread of

control. Two benefits accrue from this approach. First, we reduce the number of scheduling points by one in all cases. The second benefit is that in some cases, the task owning the entry need not exist as a separate "thread of control" at all, by can be optimized into a structure closely resembling a monitor.

We implement our approach in terms of a syntax directed translation scheme to emphasize that the translation is automatic, and driven solely by the syntax of the language. That is, each production of the language will cause certain fixed code patterns to be generated, which can then be subject to further optimization.

In the next section, we discuss the translation scheme for accept and select statements, and entry calls. In section 3 we discuss issues relating to data access and exception handling, and in section 4 we discuss some possible optimizations of the given translation schemes. In section 5 we discuss some issues relating to monitors.

2.0 GENERAL TASKING TRANSLATION SCHEMES

2.1 Compiler Generated Data Structures

Several pieces of information are associated with a task activation for the purpose of managing a rendezvous. Each task activation is described by a task activation variable that includes the following three components:

tv.gate - A boolean array representing the set of entries associated with a task. An entry can be accepted to initiate a rendezvous only if the entry is present in the gate.

tv.mutex - A semaphore which is used to control the critical sections which regulate rendezvous initiation and termination.

tv.tasksem - A semaphore used to control a task when it calls another task and has to wait for completion of the entry call.

In addition to a single task activation variable, there is an entry variable associated with each entry of a task activation. Each entry variable has two components:

ev.address - This is the address of the body of code textually associated with the currently executing accept statement. This is necessary because a task can have multiple accept statements for a particular entry.

ev.wlist - This is a list of calling tasks awaiting execution of an accept statement. It is a property of the language that a calling task can be waiting on at most one ev.wlist.

In addition, there is a semaphore associated with the code of each select statement, and a semaphore associated with the code of each accept statement that is not a select alternative. These semaphores control the execution of nested accept or select statements and intermediate code between these statements.

The compiler generates initialization code for the task's activation variable and its entry variables, as well as the accept and select semaphores when the task is initiated. The mutex semaphore is set to true, the tasksem semaphores are set to false, and each entry's waiting list is set to empty. The accept and select semaphores are set to false.

Entries can have multiple accept bodies, and as statements, accept statements can be nested within other accept statements. Consider the following example:

```
accept A(...) do
        ...
        accept A(...) do ... end A;
        accept B(...) do ... end B;
        ...
end A;

accept B(...) do ... end B;
accept C(...) do ... end C;
```

The body address of entry C is fixed, but those of entries A and B depend on the current locus of control. This explains the need for the address field in the entry variable.

2.2 An Example

In order to motivate the detailed solution that follows, it will be informative to examine the translation of an Ada task which buffers characters in an attempt to smooth variations in speed between a consuming task and a producing task [Ichbiah]. It is recommended that this example is first read to get a general impression of the issues. It is not necessary for the reader to fully understand the details of the translation as given, but rather its general shape. This example can be understood in full detail after reading section 4. It is discussed in section 4.7.

The buffering task contains an internal pool of characters processed in a round-robin fashion. The pool has two indices, an IN_INDEX denoting the space for the next input character and an OUT_INDEX denoting the space for the next output character.

```
task BUFFER is

    entry READ (C : out CHARACTER);

    entry WRITE (C : in CHARACTER);

end;


task body BUFFER is

    POOL_SIZE : constant INTEGER := 100;
    POOL : array (1 .. POOL_SIZE) of CHARACTER;
    COUNT : INTEGER range 0 .. POOL_SIZE := 0;
    IN_INDEX, OUT_INDEX : INTEGER range 1 .. POOL_SIZE := 1;

begin

    loop

        select
            when COUNT < POOL_SIZE =>
                accept WRITE (C : CHARACTER) do
                    POOL (IN_INDEX) := C;
                end;
                IN_INDEX := IN_INDEX mod POOL_SIZE + 1;
                COUNT := COUNT + 1;
        or
            when COUNT > 0 =>
                accept READ (C : CHARACTER) do
                    C := POOL (OUT_INDEX);
                end;
                OUT_INDEX := OUT_INDEX mod POOL_SIZE + 1;
                COUNT := COUNT - 1;
        end select;

    end loop;

end BUFFER;
```

The desired translation of this task will define two procedures called "read" and "write" whose execution will be mutually exclusive. In addition, there will be some initialization for the task which is executed when the task is initiated. The translation of "read" and "write" is given below:

```
initiate:
        -- allocate and initialize user variables
        -- type entry_indices is (write_index, read_index)
        -- temp_gate : array(entry_indices) of boolean

        jump to setnext;

read:   prologue(read_index);

        pool(in_index) := c;
        in_index := in_index mod pool_size + 1;
        count := count + 1;
        jump to setnext;

write:  prologue(write_index);

        c := pool(out_index);
        out_index := out_index mod pool_size + 1;
        count := count - 1;
        jump to setnext;


setnext:

        temp_gate[write_index] := count < pool_size;
        temp_gate[read_index]  := count > 0;

        p(buffer.tv.mutex);

        for each i in temp_gate loop

                if buffer.ev[i].wlist /= empty
                then

                        k := head(buffer.ev[i].wlist);
                        v(k.tasksem);
                        return;

                end if;
        end loop;

        buffer.tv.gate := temp_gate;
        v(buffer.tv.mutex);
        return;
```

```
prologue(entry) =

        p(buffer.tv.mutex);

        if not buffer.tv.gate[entry]
        then
                link me on buffer.ev[entry].wlist;
                v(buffer.tv.mutex);
                p(me.tasksem);
                unlink me from buffer.ev[entry].wlist;
        end if;

        buffer.tv.gate := empty;
        v(buffer.tv.mutex);
        return;
```

When the task is initiated, the program executes the "setnext" operation, whose purpose is to enable the acceptance of "read" if the buffer is not empty, and "write" if the buffer is not full. If any tasks are waiting on an entry queue, one is allowed to proceed. If no tasks are waiting, then the gate of "acceptable" entries is initialized, and mutual exclusion is released.

The "read" and "write" operations each are synchronized through some (nearly common) prologue code which is called, but could easily be inserted in line if desired. The purpose of the prologue code is to guarantee that the entry is disallowed from proceeding when:

- The other operation (i.e. the "buddy") is in progress.
- Another call to either read or write is starting.
- The boolean condition guarding this entry is false.

A key point is that once an operation is permitted to proceed, it does so without blocking. The calling task executes at most two P-operations on which it may block, both in the prologue. The first should usually not cause a block, as will be discussed later. The second is only invoked when the buddy operation is in progress. Thus, it is likely that the calling task will rarely block, and then, only out of necessity.

2.3 Translation Of Entry Calls

First, we describe the translation of an entry call. Then, we describe the code at the destination of the call, which we call the prologue code. This code corresponds to an entry declaration. Next, we give a default translation scheme a compiler might use to translate an accept statement which is not a select alternative. Finally, we describe the translation of a select statement.

2.4 Translation Of Entry Calls

An entry call

```
t.e(a1,a2,...,an)
```

is compiled as a simple procedure call to a piece of prologue code associated with the entry e.

```
prologue(t.tv, t.e, a1, a2, ..., an)
```

Actual parameters are associated with formal parameters in the usual way. Because the compiler cannot, in general, determine which task is calling the task that it is currently compiling, an implementation may choose to pass the caller's task variable "me" as an additional parameter to prologue.

2.5 Translation Of Entry Declarations

The prologue code is given below, in which "tv" refers to the task activation variable of the called task, "ev" is the entry variable for entry e, and "me" refers to the task activation variable of the calling task.

```
e:      p(tv.mutex);
        if not (e'index in tv.gate)
        then
                link me on ev.wlist;
                v(tv.mutex);
                p(me.tasksem);
                unlink me from ev.wlist;
        end if;
        tv.gate := empty;
        (*)                         -- Establish data context
        v(tv.mutex);
        jump indirect to ev.address;
```

This code is similar to a P-operation. It delays a calling task until it can execute the appropriate accept body. It sets the gate to empty so that no other calling task will succeed in executing an accept body after one task succeeds in passing the entrance to an accept body.

Under certain assumptions of preemptive scheduling, the operation p(tv.mutex) may be implemented as a simple busy waiting loop as no paths of execution to the corresponding v operation cause a task to be suspended with tv.mutex locked. The operation v(tv.mutex) simply sets tv.mutex to one.

If a calling task finds the entry of its choice closed, it puts itself onto the waiting queue for the desired entry and allows other tasks to execute the prologue code by performing the v(tv.mutex). The task waits on its own tasksem until awakened when another task performs a v(tv.tasksem). When the task is awakened, it does not go through another p(tv.mutex), because the running task leaves it locked. We are assured that the awakened task does not have to compete with newcomers and find that the open entry has been seized by one of them. The effect is that the running task hands over its critical section to the awakened task.

The prologue ends with a jump to the current accept body. The strategy using the entry variable's goto field guarantees that we choose the correct body. It is too early to decide which body to select when a task starts in the prologue. The calling task may have to wait and the locus of control may change before it is awakened. Therefore, the calling task does not use the ev.address field until it is selected to go ahead. The correct return address and the actual parameters have already been made available to the accept body when prologue was called.

2.6 Translation Of Accept Statements

We consider the translation of accept statements in two cases: simple accept statements which are not part of a select statement, and the accept statements of a select statement, i.e. a select alternative.

An accept statement of the form:

```
        ...

        accept e(...) do s;

        ...
```

is translated into:

```
        ...

        jump to c;
s:      code for the accept body s
        epi(acceptsem)
        return
c:      setstate(acceptsem,e,tv,ev,s);

        ...
```

Code between labels "s" and "c" is executed by the calling task, while the code starting at label "c" and the jump instruction is executed by the called task. The epi operation simply allows the called task to be activated.

```
epi(acceptsem) =

        (+)                          -- Restore data context
        v(acceptsem);
```

(The symbols marked "(*)" and "(+)" will be discussed in section 4 and are included here as points of reference.)

The setstate operation is similar to a V-operation. Its purpose is to enable other tasks to complete their prologue, and to choose one of them if they are on the waiting list.

```
setstate(acceptsem,e,tv,ev,s) =

        p(tv.mutex);
        ev.address := s;
        if ev.wlist empty
        then
                tv.gate(e) := true;
                v(tv.mutex);
        else
                choose one task (k) on ev.wlist
                v(k.tasksem);
        end if;
        p(acceptsem);
```

If there is a task waiting for an entry to open and that entry is included in the new gate value, one such task is awakened, but the critical section is not closed by a v(tv.mutex). The critical section is taken over by the awakened task at the point of the statement in the prologue which unlinks the awakened task from the waiting list. As observed earlier, this strategy prevents other tasks from overtaking the awakened task.

The operations setstate, prologue, and epi can be implemented either as closed or open code. The choice involves a space-time tradeoff that can be left up to the individual implementation.

2.7 Translation Of Select Statements

The select statement implements multi-way waiting in Ada, with optional timeouts (the delay alternative) and a provision for proceeding if no entry call has been made (the else alternative). The transformation from single accept statements to select statements is straightforward: instead of opening the gate for a simple accept body, the gate is opened for all the accept bodies given by the select alternatives.

The entry call and the prologue code are exactly as before. This is important because entries can be accepted in the same task both inside and outside of select statements. As we've observed, the binding may not be known at the time of the entry call.

Select alternatives may be guarded by Boolean expressions. The corresponding alternative can only be chosen if the guard evaluates to true. An absent guard is assumed to be true.

A select statement of the form:

```
        ...

                select
                        when b1 =>
                                accept e1 (...) do s1;
                                rest of alternative 1;
                or
                        when b2 =>
                                accept e2 (...) do s2;
                                rest of alternative 2;
                or
                                ...
                end select;

                ...
```

translates to:

```
                ...
                jump to c;
        s1:     code for s1;
                episel(selectsem,s1',labelvar);
                return
        s1':    rest of alternative 1;
                jump to c';

        s2:     code for s2;
                episel(selectsem,s2',labelvar);
                return
        s2':    rest of alternative 2;
                jump to c';

        ...

        c:      selectstate(selectsem,tv, (ev1,e1,s1), (ev2,e2,s2), ... );
                jump indirect to labelvar;
        c':
                ...
```

Selectstate is similar to setstate. It takes as arguments a semaphore on which the task executing the select waits, the task activation variable for the task executing the select, and a set of triples of entry variables, entries, and the accept bodies to which they are to be bound. There is one triple for each accept alternative in the select statement, so that the argument list is determinable at compile time.

```
selectstate(selectsem,tv,(evi,ev,si),...) =

        temp_gate := evaluate_guards(); -- returns a bit pattern
                                        -- of true guards
        p(tv.mutex);
        for each entry ei in temp_gate loop
                evi.goto := si;
                if evi.wlist not empty
                then
                        choose one entry(k) on evi.wlist;
                        v(k.tasksem);
                        jump to b;
                end if;
        end loop;

        tv.gate := temp_gate;
        v(tv.mutex);
b:      p(selectsem);


episel(selectsem,s',labelvar) =
        labelvar := s';
        (+)                             -- Restore data context
        v(selectsem);
```

2.8 Delay Alternatives

In the example shown above, the assumption was made that there is no else or delay alternative. If there is a delay alternative delaying for time dt, replace the p(selectsem) in the above example with the code that follows. If there is more than one delay alternative, the one with the smallest delay must be used, and the skeleton given below must be suitably modified. A bit is allocated in the guard which coressponds to the delay alternative. This bit is set or not depending on the evaluation of the guard for the delay alternative, in the same way as the guards for the other alternatives.

```
b:      hibernate(dt);
        p(tv.mutex);
        if tv.gate not empty
        then
                tv.gate := empty;
                v(tv.mutex);
                code the body of the delay alternative here;
        else
                v(tv.mutex);
                p(selectsem);
        end if;
```

In addition, the accept alternatives given above must be modified so that the hibernate is cancelled whenever the accept statement is executed. That is, place the following line in front of each accept body:

```
cancel_hibernate;
```

The actual details of this are system specific and are not relevent to the general scheme. Operation of a delay alternative proceeds by hibernating after all appropriate gates have been opened. If one of the accept alternatives has been executed, or the specified time has elapsed, the hibernating task is awakened, and in the former case closes the gate and executes the delay alternative body, and in the latter it simply performs a P-operation that corresponds to the V-operation executed by the calling task. (This P-operation can be optimized away, depending on how these semephores are actually implemented.)

2.9 Translation Of Else Alternatives

If there is an else alternative, then the task executing the accept should not wait at all, and so we replace the code following the loop in the definition of selectstate with the following:

```
            v(tv.mutex);
            code for the else alternative;
    b:      goto c';
```

2.10 Nested Accept Statements

Care must be taken in performing a setstate operation for a nested accept. The problem arises because the accept body for the outer accept is being executed by a task (t1) other than the one which owns the entry (t0), yet t1 has access to t0's task activation variable. Thus, if t2 performs a rendezvous with t1's execution of the inner accept, t2 must not release the semaphore on which t0 is blocked.

Fortunately, our scheme above handles this case with no modification. All that is necessary is to associate a different semaphore with the nested accept. Only one semaphore is needed for each (dynamic) nesting level of accepts, but because of procedure calls it may be easier to logically associate a different semaphore with each accept or select, and use other facilities in the compiler to perform temporal lifetime analysis on these semaphores in an attempt to merge them.

2.11 Multiple Opposing Entries

We mentioned before that an entry may have a number of different accept bodies associated with it. We deal with these through the entry variable goto field. In terms of the task body, it means that several accept bodies may have the same name. There is one case we purposely don't handle as described in the Ada reference manual. This is the case of a select statement containing several accept bodies with the same name.

```
select
        accept a(...) do ... end a;
or
        accept b(...) do ... end b;
...
or
        accept a(...) do ... end a;
end select;
```

The reference manual says that one of the a bodies must be chosen randomly. In our opinion this construct should not be allowed because of its ambiguity. In addition, if allowed, we don't like to use a random number generator. Our implementation happens to pick the last body. It can easily be changed to pick the first, but we do not really care about this.

3.0 DATA ACCESS AND EXCEPTION HANDLING

The calling task executes a portion of the called task, i.e. the accept body, as part of its own thread of control. However, the body of the accept may require access to the state space local to the called task. Because of this, and because the language requires that exceptions raised during the rendezvous are propagated in both the calling and the called tasks, it is necessary to perform some administration before execution of the accept body.

This proceeds in three steps. First, the dynamic link (frame pointer) of the called task is obtained from its saved state. This is easily accessible to the calling task as it is a property of the task activation, and the task must be known or else it couldn't be called. Second, the calling task's dynamic link is stored in a reserved location in the stack frame. This frame is marked to indicate that, if an exception occurs, it must be propagated past this frame along both dynamic links. Finally, the called task's dynamic link is stored as the dynamic link of the calling task.

These steps are executed in the code sequences given above, in the places marked by (*). The dynamic link must be restored at the end of the rendezvous, and this point is indicated above by (+).

This model assumes all tasks are running in the same address space. The language requires that some provision be made for accessing shared data across tasks, and thus if an implementation decision is made to run tasks in separate address spaces, some mechanism for this kind of data access is already (partially) present. Fortunately, parameter passing copy semantics alleviates some problems of data sharing during intertask communication.

4.0 OPTIMIZATION

The general scheme presented is quite efficient. However, some important cases allow substantial optimizations on the general scheme.

4.1 Semaphores

As mentioned earlier, under certain assumptions the mutex semaphore that controls initiation and termination operations of a rendezvous can be implemented with busy waiting on a bit of information. The justification is by examination of the code sequence. There is no opportunity for the running task to block before it releases the semaphore, and so any task waiting on the semaphore will, as far as its concerned, not have to wait very long. The key assumption that must be satisfied to implement the busy wait scheme is that the busy waiting will not block a higher priority task from executing. This can happen using preemptive priority schedulers in the following situation: task T1 is executing the prologue code and therefore is in the critical region guarded by the mutex semaphore. Task T2 preempts T1 and tries to call the same entry that T1 was calling, immediately causing deadlock.

Another optimization comes from the observation that all task semaphores are "private" in that no other task will ever perform a P-operation on such a semaphore. This means that none of the private semaphores can ever have more than a single task waiting on them, and it is always the same task. Thus, there need be no queue manipulation on a P-operation on any of these semaphores, although when unavailable, a scheduling decision may be made.

A third point to be noticed, is that a compiler may be able to observe that the lifetimes of certain semaphores do not overlap, and therefore their storage can be shared.

4.2 Special Instructions

Some systems have special instructions which operate efficiently on queues. The particular queueing strategy to be used here for entry variable waiting lists has not been specified, and is left to the compiler implementation to exploit these instructions.

4.3 Context Mapping

In some cases, accept bodies are such that no data context needs to be accessed, and so one would like to verify that the operations specified for dynamic links can be avoided. Unfortunately, in order to omit the dynamic link operations, one must verify that exceptions cannot be propagated from the calling task. This may be difficult to verify.

4.4 Non-Synonymous Accept Bodies

In the case that a compiler can determine that there is only a single accept body associated with an entry, the goto field of the entry variable can be omitted, and so can its initialization. The entry prologue code then jumps directly to the body, instead of indirectly through the ev.address. Even better, the prologue may physically precede the accept body, saving even the transfer of control.

4.5 Code Motion (A Very Important Optimization)

Counter to what is said in the reference manual, it may be advantageous to pull simple pieces of code into accept statements and avoid having code in between accept statements. In this way, the calling task executes more of the called task then just the accept body. If an intermediate piece of code can be absorbed by an accept statement, the V, P, and the invocation of setstate is placed at the end of the accept statement. The V and P-operations cancel each other, and can be removed. The branch to the P-operation is retargeted in a straightforward way.

4.6 Avoidance Of Context Creation

In extremely simple tasks, it is possible to avoid setting up a separate data context. Variables "own" to a task can be stored in the context of the task which is the static parent of the simple task.

This kind of task is commonly referred to as a "monitor". It is not clear how to characterize monitors exactly so that no separate date context is needed. Creation and placement of local variables of the task depend on code generation style, so specific guidelines cannot be given. In the simplest case, this optimization may be applied when there is no code outside of accept bodies, no procedure calls inside the accept bodies, and all variables can be allocated either in registers or in an outer context.

4.7 Transformations Used In The Buffering Example

In section 2.2 we presented the desired translation of an example of a buffering task. We now indicate how the general translation scheme of section 2 can be optimized to the desired result. The first translation moved some code and the setstate operation "into" the rendezvous (see 4.5), since these were the next operations executed outside of the select statement. It was observed that the code for the setstate operation could easily be shared by both select alternatives and the initiation code.

The second transformation is based on the observation that read and write each have only one accept statement (see 4.4), and so there is no need to use an indirect branch from the prologue code to the accept body. In this case, we chose to code the call to the prologue code directly in front of the accept body.

Finally, we were able to omit the selectsem semaphore and its V- and P-operations (see 4.6), because we could guarantee that they would always be executed in succession by the same task. This left no semaphores associated with BUFFER's thread of control. The task has now effectively been converted to a monitor.

5.0 A NOTE ON MONITORS

It has been argued [Hilfinger] that it may be useful for programmers to distinguish between monitors and processes. A monitor is a task without private code (i.e. without code sections between accept and/or select statements), while a process does have private code. We feel that the distinction has some merits, but that it should not be reflected in the syntax of Ada. There is no need to replace the keyword task by two different ones that distinguish between process and monitor. We argue that is is preferable to have a uniform task concept as currently provided in Ada.

The implementation of monitors is somewhat more efficient than that of processes. However, there is no point in expressing that the more efficient implementation is applicable by using a special keyword, because we showed that an Ada compiler will automatically generate the more efficient code if a task has no private code.

The distinction of having or not having private code is not so much related to the monitor concept, but more related to the desired concurrency in the computation. Let <p> and <q> be two code sections that can be coded either as

```
accept A(...) do <p>;  <q>;  end A;  ...
```

or as

```
accept A(...) do <p>;  end A;  <q>;  ...
```

In the first case the calling task resumes its main program <m> when both <p> and <q> have been executed. In the second case the calling task resumes <m> after <p> has been executed, while <q> can be executed in parallel with the continuation of <m>. It is clear that it depends on the nature of code <q> which alternative is more desirable. If <q> may interfere with <m> or if <q> is trivial code, the first alternative is the better one. If <q> does not interfere with <m> and is relatively long, it may be worth the extra task activation to allow for concurrent execution. The flexibility is in the choice a programmer has in writing a task body. A special keyword is not helful.

6.0 CONCLUSION

We have presented a general and flexible scheme for syntax directed translation of tasking constructs in Ada. This scheme avoids excessive waiting of tasks wishing to call entries, and may optimize tasks which own entries "out of existence". The scheme we have proposed requires no searching of queues, and can be implemented with a very high degree of efficiency on a wide range of architectures.

7.0 REFERENCES

[Hilfinger] The Tasking Facility in Ada, P.N. Hilfinger, June 1979

[Habermann] Implementation of Regular Path Expressions, CMU Technical Report ANH 7902, January 1979

[Ichbiah] Preliminary Ada Reference Manual, J. Ichbiah et al, SIGPLAN Notices, V14 N2 (June 1979).

[end of paper]