SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 11, 1019–1051 (1981)

Task Management in Ada—A Critical Evaluation for Real-Time Multiprocessors

ERIC S. ROBERTS, ARTHUR EVANS JR. AND C. ROBERT MORGAN
Bolt Beranek and Newman Inc.

AND

EDMUND M. CLARKE
Harvard Univeristy

SUMMARY

As the cost of processor hardware declines, multiprocessor architectures become increasingly cost-effective and represent an important area for future research. In order to exploit the full potential of multiprocessors, however, it is necessary to understand how to design software which can make effective use of the available parallelism. This paper considers the impact of multiprocessor architecture on the design of high-level programming languages and, in particular, evaluates the language Ada in the light of the special requirements of real-time multiprocessor systems. We conclude that Ada does not, as currently designed, meet the needs for real-time embedded systems.

1. INTRODUCTION

The possibility of using multiprocessor architecture as the basis for a powerful computing system is an attractive one for several reasons. First, a multiprocessor system can achieve significantly increased computational speed by allowing parallelism in its task structure. Second, multiprocessor architecture offers the potential for achieving system reliability through the redundancy of its processing elements. Third, as the cost of processing components (particularly the LSI-based microprocessor) declines, the cost of adding processors to a system becomes less significant in relation to the overall system cost. In the light of these advantages, interest in multiprocessor architectures has grown significantly over the past decade, and it is clear that multiprocessors are likely to become increasingly cost-effective in years to come. It is also clear that the use of multiprocessor technology has an effect on software methodology which must be considered in the design of any programming language system intended for use in a real-time multiprocessor-based environment.

In this paper, we are particularly concerned with the impact of multiprocessors on the programming language Ada[1] which was developed by Cii Honeywell Bull for the U.S. Department of Defense and is intended to serve as a programming standard for *embedded computer applications* (i.e. command and control, communications, avionics, shipboard applications, etc). As a consequence of its projected application domain, the language contains facilities for parallel and real-time programming in addition to the usual control and data structuring facilities of conventional languages such as Pascal.

0038–0644/81/101019–33$03.30

Received 2 January 1980
Revised 5 May 1980

Moreover, since Ada is designed to be used in applications which are well-suited to multiprocessor systems, we believe that a study of Ada from the point of view of multiprocessor systems is a particularly relevant area of research.

Section 2 of this paper examines the nature of a typical multiprocessor system and briefly outlines various differences between applications which are well adapted to multiprocessors and those intended for more conventional uniprocessor architectures. Secion 3 outlines the parallel control facility provided by Ada and provides a framework for a more detailed discussion of the implications of that design. Section 4 presents an evaluation of Ada's parallel processing facility giving special consideration to the unique requirements imposed by multiprocessor systems. To the extent that the primitives provided by Ada are judged to be inadequate for use in the environment of a practical multiprocessor, alternative structures and extended facilities which would relieve the major problems are discussed. These are presented as general conclusions in Section 5.

We recognize that the Ada language is still under development and that the language definition reported in Reference 1 must be viewed as a preliminary document. In fact, there are indications that changes are being incorporated into the Ada design which solve some the problems addressed in this paper. We believe that it is important to bring some of the questions related to parallel control in Ada before a wider audience and that the results presented in this paper will be of use in evaluating the designs of tasking mechanisms in general, even if the specific critique of Ada becomes dated through changes in the language definition.

2. THE NATURE OF MULTIPROCESSOR APPLICATIONS

As part of our evaluation of Ada as a language for multiprocessors, we feel that it is important to consider not only the characteristics of multiprocessors themselves but also the nature of the applications which are typically encountered in a multiprocessor environment. From our experience with existing multiprocessor systems, we believe that multiprocessor applications tend to have much more stringent requirements for run-time efficiency than do most applications developed for uniprocessor environment. This increased requirement for efficiency arises, in part, from the observation that it is considerably more difficult to design software for a multiprocessor system than for a more traditional uniprocessor. To a large extent this increase in difficulty is related to the fact that multiprocessors represent a relatively new form of system architecture. When compared to the experience which has been assembled for single processor systems and sequential algorithms, very little is known about the problems involved in multiprocessor design and parallel programming.

The fundamental implication of the increased difficulty in software development is quite simple: multiprocessor systems will rarely be used for practical applications unless the use of a multiprocessor is required by the constraints of the application. Multiprocessors have significant advantages over conventional uniprocessors in three distinct areas:

1. Multiprocessors are capable of increased effective throughput because they allow independent tasks within the application to operate in parallel.
2. Multiprocessors can be designed to include software reliability structures which exploit the inherent redundancy in the hardware to dynamically alter the system configuration in response to hardware failures.

3. Multiprocessors can be expanded gracefully as the requirements of the application change.

Of the three factors above, the need to provide efficiency through parallelism has, in our experience, proven to be the most important. Applications chosen for use with multiprocessors tend, therefore, to have (1) strict requirements for run-time efficiency and (2) highly parallel internal task structures which permit them to take advantage of the multiprocessor design.

The need to produce highly efficient code is well understood by those who have experience in designing real-time applications and is reflected in the technical requirements for a common high order language which directed the development of Ada. Section 1D of the Steelman requirements[2] specifies that language 'features should be chosen to have a simple and efficient representation in many object machines'. Moreover, Steelman recognizes that the tasking facility is particularly subject to such efficiency considerations in its requirement (Section 9B) that the 'parallel processing facility shall be designed to minimize execution time and space'.

We believe that concern for efficiency leads to the following general conclusions:

1. The use of constructs which have no efficient representation must not be required by the language design.
2. If two different constructs display a significant variation in their efficiency depending on the application environment, both should be supplied in order to provide maximum flexibility and allow the programmer to achieve the required level of efficiency.
3. Low-level facilities must be provided to achieve higher levels of efficiency than are attainable with any general mechanism.

It is important to note that the impact on overall efficiency from the use of an inappropriate mechanism for parallel control can be extremely high when compared to the efficiency cost generally associated with programming in a high-level language. While the techniques available for optimizing serial code are highly developed and quite successful in practice, relatively little is known about the problem of optimizing the global task structure and the internal synchronization process. Based on our experience with multiprocessor systems, we believe that these problems are extremely hard and well beyond the current state of software technology. This fact increases the importance of allowing greater flexibility in the task structure than might be required in the non-parallel aspects of a language.

3. AN OVERVIEW OF PROCESS CONTROL IN ADA

In this section, we present a brief overview of the parallel processing facilities in the Ada programming language to provide a background for the evaluation of those features presented in Section 4. For the most part, we have not attempted to cover the structure of the language in its entirety and have chosen to concentrate on the tasking facility alone. In the light of the similarity between Ada and conventional programming languages such as Pascal, the reader should have no difficulty following the examples of this section in spite of the absence of a full description of the language. To increase the readability of the examples, we have attempted to write code so as to

maximize readability and to use comments whenever the intent of the code might be unclear. In Ada, comments are introduced by '- -' and extend to the end of the line.

3.1. The general structure of parallel tasks

Ada uses the term 'task' to refer to the basic syntactic unit for process definition. A *task* consists of two parts: a *specification part* which describes the external behaviour of the task, and a *task body* which describes its internal behaviour. The specification part consists of a header which gives the name of the task and a declarative part which describes those features of the task which are visible to the outside world. Included in the declarative part are the declarations of those constants, types, subprograms, exceptions, and entries which are associated with the task and must be externally visible.

An example of a task specification is shown below:

```
task BUFFER is
  PACKET_SIZE : constant INTEGER : = 256;
  type PACKET is array (1 . . PACKET_SIZE) OF CHARACTER;
  entry READ (V : out PACKET);
  entry WRITE (E : in PACKET);
end BUFFER;
```

Entries are used for communication between tasks and look externally like procedures.

The task body consists of a declarative part which describes local data structures and a sequence of statements which implement the *entry* declarations described in the specification part. For the BUFFER example the task body is:

```
task body BUFFER is
  BUFSIZE : constant integer : = 10;
  BUF     : array (1 . . BUFSIZE) of PACKET;
  IN, OUT : INTEGER range 1 . . BUFSIZE : = 1;
  COUNT   : INTEGER range 1 . . BUFSIZE : = 0;
begin
  - - statements for entries READ and WRITE - -
end BUFFER;
```

The statements implementing the buffer operations READ and WRITE are given later in the chapter after additonal features of Ada have been described.

The term 'thread of control' is used to describe the execution of a task. When a thread of control enters a scope containing task declarations, the elaboration of each declaration creates a new potential thread of control. The *parent* of a task is the task whose thread of control eleaborates the task declaration. In order to cause the task body to be executed, the task name must be explicitly named in an *initiate* statement, e.g.

```
initiate PRODUCER, CONSUMER, BUFFER;
```

The tasks named in the initiate statement are activated and run in parallel with each other and with the parent task. Note that the parent of a task may be different from the

task which initiated it, although both must have access to the task's name. Consider:

```
task body T1 is
  task T2 is
    ...
  end T2;
  task body T2 is
    ...
  end T2;
  task T3 is
    ...
  end T3;
  task body T3 is
    ...
    initiate T2;
    ...
  end T3;
begin
  ...
  initiate T3;
end T1;
```

Here, T1 is the parent of T2, but T2 was initiated by T3 instead of T1.

Normal termination of a task occurs when control reaches the end of the task body. If the terminating task is a parent, then it may have to be delayed until all of its offspring have terminated. Tasks may also be terminated by means of an explicit *abort* statement. For example, the statement

```
abort T1, T2;
```

causes tasks T1 and T2 plus any descendent tasks to be terminated unconditionally. In this case a TASKING_ERROR exception is raised in those tasks which were communicating with the aborted task or its descendents.

Facilities are also provided for determining the status of a task. The system attribute T'PRIORITY may be used to determine the priority that has been assigned to task T by the scheduling algorithm which allocates available processors to tasks. The priority of a task may be changed by means of a call on the procedure SET_PRIORITY to reflect a change in the urgency of process execution.

Ada also provides arrays of tasks called *task families* to handle those situations in which it is necessary to construct a large number of similar tasks. A typical use for task families occurs when there are multiple copies of some physical device such as a console terminal and a distinct copy of the same task is necessary to drive each device, i.e.

```
task TELETYPE_DRIVER (1 .. 100) is
  type LINE is array (1 .. 132) of CHARACTER;
  entry WRITELINE (TEXT : in LINE);
  entry READLINE (TEXT : out LINE);
end TELETYPE_DRIVER;
```

```
task body TELETYPE_DRIVER is
  - - statements to implement WRITELINE and READLINE - -
end TELETYPE_DRIVER;
```

Individual copies of the task may be referred to by appending the appropriate subscript to the task name. Thus the statement

```
initiate TELETYPE_DRIVER (3);
```

will cause the third copy of task TELETYPE_DRIVER to become active.

Storage for tasks may be allocated either when the task declaration is elaborated (static creation) or when the task is initiated (dynamic creation). The choice between static allocation and dynamic allocation is determined at compile time by the use of a *pragma* or translator command, e.g.

```
pragma CREATION (STATIC);
pragma CREATION (DYNAMIC);
```

Dynamic creation is particularly important for task families where the index range provides an upper bound on the number of active processes and storage might be wasted if all tasks were allocated at the same time.

3.2 Entry declarations and the ACCEPT statement

Communication between tasks is provided by *entry* calls and *accept* statements. When one task needs to communicate with another task, it executes an *entry call*. *Entry* calls specify the information to be exchanged between the tasks and have exactly the same form as procedure calls. Thus in the bounded buffer example from the last section, a producer task places data in the buffer by executing the *entry* call

```
BUFFER . WRITE (PRODUCER_DATA);
```

and a consumer task executes the call

```
BUFFER . READ (CONSUMER_DATA);
```

to retrieve data from the buffer.

In order for an *entry* call to be syntactically correct, the called task must contain an *entry declaration* with a corresponding name and formal part. *Entry* declarations resemble procedure declarations and contain information about the type and mode of the formal parameters of the *entry*. An *entry* declaration can also specify an array or family of entries all of which have the same name and parameters. In this case, subscripts must be used to distinguish a particular *entry* in the family. Thus, in a disk head scheduler it may be convenient to associate a distinct *entry* with each track on the disk

```
entry TRANSFER (1 . . 200) (D : DATA)
```

When another task wishes to write on track I, it issues an *entry* call of the form

```
TRANSFER (I) (DATA_REQUEST);
```

The *accept* statement is analogous to the body of a procedure and indicates to the called task which statements should be executed when a particular *entry* call occurs. The formal part of the *entry* declaration is repeated at the beginning of the *accept* statement in order to emphasize the scope of the *entry* parameters. Following the formal part are the statements to be executed when the entry call is accepted. The *accept* statements for the entries READ and WRITE in the bounded buffer example are

shown below:

```
accept WRITE (E : in PACKET) do
  BUF (INX) : = E;
end WRITE;
accept READ (V : out PACKET) do
  V : = BUF (OUTX);
end READ;
```

The variables INX and OUTX are integers which point, respectively, to the rear and the front of the buffer and are declared in the body of the task (the complete example is presented later in this section). It is important to note that these variables need not be incremented within the *accept* statements. Since *accept* statements are executed in mutual exclusion, it is important for them to be as short as possible and not contain unnecessary statements. *Accept* statements for *entry* families must be subscripted to distinguish different entries in the same family. Thus, *accept* statements for the disk head scheduler example will typically have the form

```
accept TRANSFER (D : in DATA ) do ... end TRANSFER;
```

The synchronization between the calling task and the called task in an *entry* call is similar to the *rendezvous* that occurs with Hoare's CSP language.[3] As in Hoare's language there are two possibilities for a rendezvous, depending on whether the calling task issues the *entry* call before or after the corresponding *accept* statement is reached by the called task. In either case the process which reaches the rendezvous first is delayed until the other process has an opportunity to catch up. When the rendezvous is achieved, the *in* parameters of the *entry* call are passed to the called task. The calling task is then suspended while the called task executes the body of the *accept* statement. After execution of the *accept* statement, the values of *out* parameters are passed back to the calling task, and the two tasks are allowed to proceed independently again. A queue of waiting tasks is associated with each *entry* to handle those situations in which several different tasks simultaneously access the same *entry*. Tasks are removed from the queues in a FIFO manner each time that a rendezvous occurs. Note that the naming problem which occurs in Hoare's language is avoided by Ada since it is unnecessary for a called process to know the name of the calling process.

3.3. The select statement

Many of the disadvantages of semaphores stem from lack of control over what happens when a semaphore is found to be busy. Thus, it is not possible to program an alternative action to be executed when a semaphore is busy nor is it possible to wait for one of several semaphores to be free. The *select* statement in Ada provides a mechanism for avoiding this type of problem. Syntactically, the *select* statement resembles a case statement in which each alternative is a conditional statement:

```
select
  when B1 ⇒ A1;
  or when B2 ⇒ A2;
  ...
  or when BN ⇒ AN;
  else S;
end select;
```

Each *when* condition may contain an arbitrary boolean expression involving variables which are visible to the task and may be omitted if the condition is known to be true. The *select* alternatives A1, ..., A*N* are sequences of statements in which the first statement is always an *accept* statement or a delay statement. The *else* clause is simply a sequence of statements and can also be omitted if the guarding conditions B1, ..., B*N* are mutually exhaustive. A *select* alternative is said to be *open* if there is no preceding *when* clause or if the corresponding condition is true; otherwise it is said to be *closed.*

The execution of a *select* statement is described by the following five rules.

1. All of the conditions are evaluated to determine which alternatives are open.
2. An open alternative starting with an *accept* statement may be executed if the corresponding rendezvous is possible.
3. An open alternative starting with a delay statement may be executed if no other alternative has been selected before the specified time interval has elapsed.
4. If no alternative statement can be immediately selected and there is an *else* clause, then the *else* clause is executed next. If there is no *else* clause, the task waits until an open alternative can be selected by rule 2 or rule 3.
5. If all alternatives are closed and there is an *else* clause, the *else* part is executed. If there is no *else* clause, the exception SELECT_ERROR is raised.

With the *select* statement we can now complete the task body in the bounded buffer example:

```
task body BUFFER is
  SIZE        : constant INTEGER : = 10;
  BUF         : array (1 .. SIZE) of PACKET;
  INX, OUTX   : INTEGER range 1 .. SIZE : = 1;
  COUNT       : INTEGER range 0 .. SIZE : = 0;
begin
  loop
    select
      when COUNT < SIZE ⇒
        accept WRITE (E : in PACKET) do
          BUF (INX) : = E;
        end WRITE;
        INX : = INX mod SIZE + 1;
        COUNT : = COUNT + 1;
    or when COUNT > 0 ⇒
        accept READ (V: out PACKET) do
          V : = BUF (OUTX);
        end READ;
        OUTX : = OUTX mod SIZE + 1;
        COUNT : = COUNT - 1;
    end select;
  end loop;
end BUFFER;
```

The buffer is represented by a circular array with the variables INX and OUTX indicating the portion of the array which contains data. The guard COUNT < SIZE in the

first alternative of the *select* statement protects the buffer from overflow during the execution of a write operation. Similary, the guard COUNT > 0 in the second alternative protects the buffer from underflow during a read operation. Note that if 0 < COUNT < SIZE and both a read call and a write call occur, the *accept* statement that is selected will be chosen in a completely random manner. The programmer, therefore, must be careful that this non-determinism in the selection of alternatives does not affect the correctness of the program.

3.4. The delay statement, interrupts and generic tasks

In this section we describe three additional process control features provided by Ada. These features do not affect the expressive power of the language as significantly as the features discussed previously and are therefore not described in as great detail.

The first feature is the *delay* statement which can be used to postpone execution of a task for a specified interval of time. The delay statement has the form

delay ⟨simple expression⟩

The expression following the delay statement represents the length of time (in units of the real time clock) that the process is to be delayed. A delay statement can be used in place of an *accept* statement in an alternative of a *select* statement. In this case if no rendezvous occurs during the specified time interval, the statement list following the delay statement will be executed. Thus, an additional alternative of the form

or delay 10.0*MINUTES ; initiate SYSTEM_TEST;

may be added to the *select* statement in the task body for the bounded buffer example. This modification will cause the diagnostic task SYSTEM_TEST to be run if a ten minute time interval passes in which there are no READ or WRITE *entry* calls.

The second feature is the *interrupt entry*: in Ada, hardware interrupts are simply interpreted as external *entry* calls. An Ada *representation specification* is used to link the *entry* to the physical storage address which records the interrupt. The interrupt is processed exactly the same way that any other *entry* call is processed; thus, the queuing mechansim for *entry* calls can be used to handle multiple interrupts. Likewise, the mechanism for masking interrupts can be hidden from users by incorporating it in the software which connects the interrupts to the *entry* call. To illustrate how interrupts are handled in Ada, we show how a *stop* button can be added to the bounded buffer example. We assume the existence of a console button which can be pressed to cause a hardware interrupt. A representation specification of the form

for STOP use at 8#7777;

can be used to associate the *entry* STOP with the physical address of the interrupt. If the *select* statement in the task body is modified to include the alternative

or accept STOP; exit;

then loop will be terminated when the stop button is pressed.

The final process control feature that we discuss is the *generic task*. The bounded buffer example described earlier in this chapter does not provide users with a general mechanism for declaring buffer tasks. By making the tasks generic, i.e. by changing

the specification part of the task to

```
generic task BUFFER is
  PACKET_SIZE : constant INTEGER : = 256;
  type PACKET is array (1 . . PACKET_SIZE) of CHARACTER;
  entry READ (V : out PACKET);
  entry WRITE (E : in PACKET);
end
```

this difficulty can be overcome. When a user needs to declare a new instance of a bounded buffer, the construction

```
task BB is new BUFFER;
```

may be used. READ and WRITE calls on the new instance of the bounded buffer have the syntax:

```
BB . WRITE (PRODUCER_DATA);
BB . READ (CONSUMER_DATA);
```

Signals and *semaphores* are provided by Ada as predefined generic tasks. If Ada is implemented on a machine on which these primitives are provided by hardware, then the compiler can directly translate *entry* calls into the corresponding hardware primitives. In doing so, however, it is critical that the semantics of the language remain entirely unchanged. As noted in sub-section 4.2.3., the FIFO semantics of the Ada rendezvous can make this particularly difficult to achieve.

4. EVALUATION OF PROCESS CONTROL IN ADA

As discussed in Section 2, we believe that the use of multiprocessor systems tends to be most valuable in those applications in which run-time efficiency is a critical concern. For this reason, we feel that the parallel control features provided by an implementation language intended for use with multiprocessors must be designed to allow highly efficient interprocess communication and control. After reviewing the Ada language in detail, we are concerned that the primitives provided by Ada do not allow the programmer to achieve this desired level of efficiency. Furthermore, in order to avoid the efficiency cost associated with the Ada task structure, programmers will be forced to adopt an unnatural coding discipline that will make programs more difficult to read and understand.

4.1. Scheduling and the rendezvous

The most severe problem with the process control features in Ada from the point of view of efficiency is that the transmission of data from a sender process to a receiving process requires excessive scheduler interactions. Our experience is that message passing of this type occurs frequently in real-time applications, and that in such applications it is necessary to reduce the number of interactions with the schuduler to a minimum to meet the relevant time constraints.

4.1.1. An example of scheduling delay

To illustrate the problem, we examine the problem of passing messages from a sender process to a receiving process where no response or acknowledgment is required. Conceptually, we imagine that there is a queue linking the sender and receiver which can hold some finite number of messages in transit. When the sender

process generates a message, it enters the associated data at the end of the queue. The receiver process, whenever it is free to accept a new message, simply takes the first message from the queue. In a parallel environment, it is desirable that the sending operation (i.e. entering the data on the queue) be performed without incurring any significant delay so that the sending process can continue its operation as quickly as possible. In particular, when the queue is not full, there should be no required scheduler interactions.

Consider the bounded buffer example presented in sub-section 3.1. This example has been used to demonstrate that bufffered message passing with non-blocking senders can be implemented in Ada. If *entry* calls are implemented as described in the *Ada Rationale* [Reference 1, page 11–40], however, the delay arising from scheduler actions seems extremely severe and impossible to avoid. Consider, for example, the scheduler interactions involved when a producer task sends a packet of data to a consumer task. Assume that the producer task executes the *entry* call

```
BUFFER . WRITE (PRODUCER_DATA);
```

to initiate the transfer. Given the semantics of the *entry* call, the producer is now blocked until the buffer task is scheduled and completes the rendezvous. During this time, the producer process is suspended and must wait to be rescheduled when the buffer task completes. Thus, before the producer is allowed to continue, two scheduling operations must occur. Furthermore, the implementation discussion in the *Ada Rationale* indicates that the buffer task should dismiss after completing the rendezvous in order to allow tasks of higher priority to run at that point, so that it will not immediately be able to perform a rendezvous with a consumer process.

Essentially the same sequence of operations is performed when the consumer task executes the corresponding *entry* call

```
BUFFER . READ (CONSUMER_DATA);
```

to receive a message. This implies that a total of four scheduling interactions is required to transmit a single message. Since each scheduler interaction may involve a complete context swap, this implementation of message passing would be prohibitively expensive for many applications.

Note that this problem does not arise if the message passing mechanism is implemented through the use of a message queue or directly by the harware of the target machine. The queue operations themselves must be protected against concurrent updates through some mutual exclusion mechanism, but in this case it is reasonable to use interlocks or some similar mechanism based on busy waiting without incurring the overhead of a scheduler interaction. From the statistics on lock contention given in Reference 4, we see that neither the producer task nor the consumer task will be delayed for an inordinate period of time.

From our experience with real-time communications systems, it is evident that the scheduling delay above presents a serious problem that must be solved for Ada to be recognized as an acceptable implementation language for multiprocessor systems. In the search for a solution, one has two potential choices:

1. Without changing the Ada language, develop some mechanism which would permit the translator to produce more efficient code in those cases where it can be determined that the rendezvous is not necessary.
2. Add new features to Ada to support a more efficient mechanism for message passing without sender delays.

4.1.2. *The Habermann/Nassi implementation of rendezvous*

In this section, we describe a solution to the problem of scheduling delay which was developed by Habermann and Nassi and described briefly by Habermann in his commentary on the RED and GREEN candidates for the Ada language.[5] The Habermann/Nassi solution consists of replacing the *entry/accept* interface with an alternative implementation resembling a procedure call. The interesting feature of this change in implementation is that the statements in the range of the *accept* statement are evaluated, not by the called task, but by the caller. If this is done correctly, the calling task need never dismiss its processor and therefore is not forced to wait for the scheduler.

In his evaluation of the Ada tasking facility, Habermann observes that many of the tasks that arise in practical applications are of the 'server' type and consist of one or more *select* statements enclosed in a loop (the BUFFER task above is of this type). Habermann argues that tasks of this type often permit the compiler to eliminate the rendezvous by replacing the *accept* statement linkage with a subroutine which implements the required mutual exclusion and synchronization with some internal primitive such as a semaphore. He briefly outlines a scheme for performing this transformation by analysing a variety of cases. In the paragraphs below, we have attempted to reconstruct this argument in a simpler form and then apply it to the BUFFER example.

In the course of this discussion, we will need to introduce internal semaphore objects to control the program flow. Although semaphores may be implemented in Ada using task entities, we feel that it is clearer to think of these semaphores as data objects of type SEMAPHORE which have two values (LOCKED and UNLOCKED) and two primitive operations (P and V). We will therefore write semaphore operations in the more conventional functional notation (i.e. P(SEM1) instead of SEM1 . P).

Later in this section, we will also make use of a special P operation, which we will call JOINTP, which takes two semaphores and waits until *both* semaphores are in an UNLOCKED state. Note that this is not the same as a wait for one semaphore, followed by a wait for the other, since neither semaphore is actually locked until both are available.

As a simple case, consider a task whose body consists entirely of a sequence of *accept* statements in a loop such as

```
task body EXAMPLE1 is
begin
  loop
    accept ENTRY1 do
      -- ⟨body of ENTRY1⟩ --
    end ENTRY1;
    accept ENTRY2 do
      -- ⟨body of ENTRY2⟩ --
    end ENTRY2;
    -- more accept statements --
    accept ENTRYn do
      -- ⟨body of ENTRYn⟩ --
    end ENTRYn;
  end loop;
end EXAMPLE1;
```

To translate this example into its procedural equivalent, we associate each of the entries (ENTRY*i*) with an internal semaphore (SEM*i*) and translate each *accept* statement into a procedure declaration which begins by performing a P operation on its associated semaphore and ends by performing a V operation on the semaphore associated with its successor *entry*. The 'entry procedures' then have the form

```
procedure ENTRY1 is
begin
  P(SEM1);
  -- ⟨body of ENTRY1⟩ --
  V(SEM2);
end ENTRY1;
```

and so on up to

```
procedure ENTRYn is
begin
  P(SEMn);
  -- ⟨body of ENTRYn⟩ --
  V(SEM1);
end ENTRYn;
```

In this case, since no code exists in EXAMPLE1 that is not enclosed in *accept* statements, no actual thread of control need exist for EXAMPLE1 and the initiation of EXAMPLE1 consists simply of declaring the semaphores SEM1 to SEMn, with SEM1 initialized to UNLOCKED and the remaining ones in the LOCKED state. After considering the actions of the semaphores in the example above, it should be clear that the control semantics of the procedural version is identical to that of the rendezvous provided that semaphores are implemented so as to ensure the first in/first-out discipline. Initially, the 'task' will only accept *entry* calls to ENTRY1, since any other call will block on the P operation at entry. The first call to ENTRY1, on the other hand, will succeed, and the V operation at the end of the procedure body will allow the system to accept a call on ENTRY2 or to process an existing call pending on the associated semaphore.

A simple version of the *select* statement may be handled through the use of semaphores in a similar fashion. Consider, for example, the task specification below:

```
task body EXAMPLE2 is
begin
  loop
    select
      accept CASE1 do
        -- ⟨body of CASE1⟩ --
      end CASE1;
    or
      accept CASE2 do
        -- ⟨body of CASE2⟩ --
      end CASE2;
    end select;
  end loop;
end EXAMPLE2;
```

In this example, we will need to declare a semaphore with the *select* statement (SELECT_SEM) to ensure mutual exclusion of the independent entries. This task may be coded in procedure form as follows:

```
procedure CASE1 is
begin
  P(SELECT_SEM);
  -- ⟨body of CASE1⟩ --
  V(SELECT_SEM);
end CASE1;

procedure CASE2 is
begin
  P(SELECT_SEM);
  -- ⟨body of CASE2⟩ --
  V(SELECT_SEM);
end CASE2;
```

Initiation of the task EXAMPLE2 corresponds to setting the state of SELECT_SEM to UNLOCKED thus allowing the first *entry* call to succeed. In this example, the first call on either of the entries CASE1 or CASE2 will succeed and will perform the actions in the body of the associated *accept* statement in mutual exclusion because of the protection provided by the semaphore. Upon completion of the *entry* body, the semaphore will once again become free and the system may service any further calls on either of the entries. It is interesting to note that this program transformation provides for 'random' ordering in the *select* statement by implicitly implementing the 'order of arrival' method discussed in the *Ada Rationale*.

The examples presented above, however, are overly simplified in that they do not provide for code within the body of the task which is not enclosed in an *accept* statement. This case requires a slightly more complex treatment that forces the server task to maintain an independent thread of control. To illustrate the basic notion involved in this generalization, consider the simple task skeleton below:

```
task body EXAMPLE3 is
begin
  loop
    -- ⟨statement body 1⟩ --
    accept ENTRY1 do
      -- ⟨body of ENTRY1⟩ --
    end ENTRY1;
    -- ⟨statement body 2⟩ --
    accept ENTRY2 do
      -- ⟨body of ENTRY2⟩ --
    end ENTRY2;
  end loop;
end EXAMPLE3;
```

With the exception of the intervening ⟨statement body⟩ code, this task is identical in form to that given in task EXAMPLE1, and we would like to identify some similar procedural form for the bodies of the *entry* calls. This can be done by associating each of the ⟨statement body i⟩ segments with a semaphore (STATEMENT_SEM*i*) in much the same way as the *entry* semaphore association (here ENTRY*i* is associated with the semaphore ENTRY_SEM*i*). Originally, only STATEMENT_SEM1 is UNLOCKED; the remaining semaphores are initialized to the LOCKED state. The task is then divided into a component which represents the 'real' task (i.e. the code outside of the *accept* statements) and the *entry* procedures, giving rise to the code segments below:

```
task body TRANSFORMED_EXAMPLE3 is
begin
  loop
    P(STATEMENT_SEM1);
    -- ⟨statement body 1⟩ --
    V(ENTRY_SEM1);
    P(STATEMENT_SEM2);
    -- ⟨statement body 2⟩ --
    V(ENTRY_SEM2);
  end loop;
end TRANSFORMED_EXAMPLE3;

procedure ENTRY1 is
begin
  P(ENTRY_SEM1);
  -- ⟨body of ENTRY1⟩ --
  V(STATEMENT_SEM2);
end ENTRY1;
and
procedure ENTRY2 is
begin
  P(ENTRY_SEM2);
  -- ⟨body of ENTRY2⟩--
  V(STATEMENT_SEM1);
end ENTRY2;
```

In this example, each of the statement sequences enables the succeeding *entry* and vice versa, which insures the correct semantics with respect to synchronization and mutual exclusion.

Finally, we need a mechanism for managing the effect of *when* clauses appearing in the *select* statement body. In effect, the *when* clauses, taken together, can be viewed as a single code body outside of the *select* statement which evaluates each of the predicates and determines which of the entries should even be considered. Since the evaluation of these predicates takes place outside the range of *accept* statements, the use of *when* clauses implies that the server task must have a separate thread of control to execute the predicate-evaluation code. For example, suppose that we were to

modify the code for EXAMPLE2 to include *when* clauses as in the following example:

```
task body EXAMPLE4 is
begin
  loop
    select
      when PREDFN1 (...)⇒
        accept CASE1 do
          -- ⟨body of CASE1⟩ --
        end CASE1;
    or
      when PREDFN2(...)⇒
        accept CASE2 do
          -- ⟨body of CASE2⟩ --
        end CASE2;
    end select;
  end loop;
end EXAMPLE4;
```

where PREDFN1 and PREDFN2 are some form of predicate (either a function call, as here, or a logical expression) that is used to control which of the *select* clauses should be accepted. As in the previous case, we wish to transform the task body of EXAMPLE4 so that the code required to compute the predicates lies in the body of the 'real' task. We will make use of four semaphores in this example: one for each *when* clause (WHEN1 and WHEN2), one to ensure mutual exclusion for the *select* alternatives (SELECT_SEM), and one to control sequencing (STATEMENT_SEM). Of these, STATEMENT_SEM and SELECT_SEM are initialized to UNLOCKED and the two WHEN semaphores are set to a LOCKED state. The code for computing the predicate expressions is given below:

```
task body TRANSFORMED_EXAMPLE4 is
  P1, P2 : BOOLEAN;
begin
  loop
    P(STATEMENT_SEM);
    P1 : = PREDFN1(...);
    P2 : = PREDFN2(...);
    if not (P1 or P2) then
      raise SELECT_ERROR;
    end if;
    if P1 then V(WHEN1) end if;
    if P2 then V(WHEN2) end if;
  end loop;
end TRANSFORMED_EXAMPLE4;
```

The code for the two CASE entries, however, is somewhat tricky. We are tempted to write procedure-type entries of the form:

```
procedure CASE1 is
begin
  P(WHEN1);
  P(SELECT_SEM);
  -- ⟨body of CASE1⟩ --
  V(SELECT_SEM);
  V(STATEMENT_SEM)
end CASE1;
```

Unfortunately, this approach is overly simplified and does not correctly ensure that only one of the *select* alternatives is evaluated. We must take two additional precautions to ensure the correct semantics of the *select* mechanism. First of all, whenever a particular entry is evaluated, we must make it impossible for the system to accept other entry calls by locking the corresponding semaphores controlling the remaining *select* alternatives. This can be accomplished by including a statement of the form

```
WHENx : = LOCKED;
```

for each of the remaining alternatives for this *select* statement. Unfortunately, even this does not fully insure sematically correct evaluation because of the ordering constraint on the semaphore operations. Since we test the WHEN semaphores prior to testing SELECT_SEM, it is possible for both CASE1 and CASE2 to have passed the first P operation, even though one will be prohibited from continuing until the other has completed the interior region. When this process completes the body of code and performs the V(SELECT_SEM) operation, there is nothing to prohibit the other branch from executing as well, since the effect of the

```
WHENx : = LOCKED;
```

has been negated by the fact that the other thread of control has already passed the point at which this test is relevant. Changing the order of the P operations will not work, since this leaves the system susceptible to deadlock states. Moreover, it is insufficient to introduce internal flags to mark the operation, because it will be impossible to tell, in general, whether any other processes have passed the first P operation unless some other action is performed indivisibly with that call to P.

The simplest correction conceptually is to replace the individual P operations with a JOINTP operation which waits for the two semaphores to become UNLOCKED together. In this case, the code for the *entry* procedures becomes:

```
procedure CASE1 is
begin
  JOINTP(WHEN1, SELECT_SEM);
  -- ⟨body of CASE1⟩ --
  WHEN2 : = LOCKED;
  V(SELECT_SEM);
  V(STATEMENT_SEM)
end CASE1;
```

and

```
procedure CASE2 is
begin
  JOINTP(WHEN2, SELECT_SEM);
  -- ⟨body of CASE2⟩ --
  WHEN1 := LOCKED;
  V(SELECT_SEM);
  V(STATEMENT_SEM);
end CASE2;
```

This technique is adequate to solve the problem, but illustrates some of the complexity that arises in more complicated applications of the Habermann/Nassi technique.

To illustrate the power of the complete mechanism, consider the following transformation of the BUFFER task which combines the individual techniques described above. For simplicity, all statements within the range of a *select* alternative have been moved inside the corresponding *accept* statement, although the technique used in EXAMPLE3 illustrates the general method for restoring the potential concurrency.

```
package NEWBUFFER is
  PACKET_SIZE : constant INTEGER := 256;
  type PACKET is array (1 .. PACKET_SIZE) of CHARACTER;
  task NEWBUF;
  procedure READ (W : out PACKET);
  procedure WRITE (E : in PACKET);
end NEWBUFFER;


package body NEWBUFFER is
  SIZE           : constant INTEGER := 10;
  BUF            : array (1 .. SIZE) of PACKET;
  INX, OUTX      : INTEGER range 1 .. SIZE := 1;
  COUNT          : INTEGER range 0 .. SIZE := 0;
  STATEMENT_SEM  : SEMAPHORE := UNLOCKED;
  SELECT_SEM     : SEMAPHORE := UNLOCKED;
  WHEN1, WHEN2   : SEMAPHORE := LOCKED;

  task body NEWBUF is
  begin
    loop
      P(STATEMENT_SEM);
      -- given the range of COUNT at least --
      -- one of the following is true so no --
      -- SELECT_ERROR exception can occur. --
      if COUNT < SIZE then V(WHEN1) end if;
      if COUNT > 0    then V(WHEN2) end if;
    end loop;
  end NEWBUF;
```

```
    procedure WRITE (E: in PACKET) is
    begin
      JOINTP(WHEN1, SELECT_SEM);
      BUF (INX) : = E;
      INX : = INX mod SIZE + 1;
      COUNT : = COUNT + 1;
      WHEN2 : = LOCKED;
      V(SELECT_SEM);
      V(STATEMENT_SEM);
    end WRITE;

    procedure READ (W: out PACKET) is
    begin
      JOINT(WHEN2, SELECT_SEM);
      W : = BUF (OUTX);
      OUTX : = OUTX mod SIZE + 1;
      COUNT : = COUNT - 1;
      WHEN1 : = LOCKED;
      V(SELECT_SEM);
      V(STATEMENT_SEM);
    end READ;
  end NEWBUFFER;
```

From the point of view of efficiency, it is evident that the above implementation strategy is preferable to the cooperating process model of rendezvous suggested in the *Ada Rationale*, but there are some costs associated with this approach, largely in terms of the complexity this structure imposes on an otherwise simple model. In particular, the Ada semantics cannot be maintained if the body of the *accept* statement is viewed as a subroutine of the caller which communicates with the called task solely through the internal semaphore structure. The generated code must take account of the fact that two separate tasks are involved.

The complexity arises because of the 'identity crisis' which occurs for the task executing the statements within an *accept* body. In many ways, it is convenient to think of the calling and called tasks as completely distinct entities. This view is made explicit in the *Ada Rationale* (page 11–40) which emphasizes that 'the caller executes a procedure himself whereas an *accept* statement is executed by the callee on the caller's behalf'. Under the Habermann/Nassi implementation, this distinction is no longer clear since the savings in efficiency result from allowing the calling task to execute the *accept* body as a procedure call.

In some cases, the identity of the task executing the code may be of some importance. For example, to allow metering of an application program, it is important that the runtime consumed during the *accept* body be charged to the CLOCK attribute of the called task rather than its caller. It is also important to remember that exception conditions which occur during the execution of the *accept* statement must be raised in both the caller and called task. Considerations such as these indicate that some form of context switching to identify the called task must be performed as part of the *entry/accept* linkage.

We also gave several examples earlier that show that the order in which semaphores are locked is extremely important and that there are cases in which the only convenient

solution is to use a joint P operation which is capable of waiting for two semaphores to become UNLOCKED simultaneously. There are other issues that complicate this structure, such as the use of the same *entry* name in two or more *when* clauses. These problems are not unsolvable by a compiler; our principal assertion is that they are conceptually more difficult to implement than the basic queuing model of task communication, which is at least as efficient in its implementation. Thus we argue that while the Habermann/Nassi solution is not hopelessly complex, it is at least unnecessarily complex.

4.1.3. *Automatic data queuing*

An alternative approach to the problem would be to devise a queue implementation which retains the linguistic structure of the *entry/accept* linkage. Presumably, this sort of structure is meaningful only in those cases in which the flow of information is unidirectional and where the synchronization provided by the rendezvous is known to be irrelevant. When these conditions apply, it is possible to achieve a significant increase in message passing efficiency by building a data queue into the task communication structure and allowing the sender to proceed.

It is immediately evident that this type of approach changes the nature of the implementation strategy. In the implementation of the rendezvous proposed in the *Ada Rationale* or the Habermann/Nassi alternative described above, no form of data queuing is ever supported by the implementation. The only entities which are entered in queues are tasks, and each task, because of the structure of the rendezvous, may be entered on at most one queue. This is extremely convenient since it allows arbitrary queuing of tasks without encountering a memory allocation problem; it is sufficient to reserve a queue pointer cell in the activation record of each task. Data queuing, on the other hand, requires that space be available to hold each of the data items on the queue. Assuming that dynamic allocation of this queue space is unmanageable, one is required to impose an upper bound on the queue size which is fixed at translation time.

In order to illustrate the general mechanism, consider the task specification below which performs the inverse of the LINE_TO_CHAR function illustrated in the *Ada Rationale* (page 11-6).

```
task CHAR_TO_LINE is
  type LINE is array (1 .. 80) of CHARACTER;
  entry PUT_CHAR <80> (C : in CHARACTER);
  entry GET_LINE (E : in LINE);
end CHAR_TO_LINE;

task body CHAR_TO_LINE is
  BUFFER : LINE;
begin
  loop
    for I in 1 .. 80 loop
      accept PUT_CHAR (C : in CHARACTER) do
        BUFFER (I) : = C;
      end PUT_CHAR;
    end loop;
    accept GET_LINE (L : out LINE) do
```

```
        L : = BUFFER;
      end GET_LINE;
    end loop;
  end CHAR_TO_LINE;
```

Note that the syntax of the *entry* declaration has been extended to allow a queue size indicator as in

```
entry PUT_CHAR <80> (C : in CHARACTER);
```

The <80> parameter specifies a queue size for communication between the callers of PUT_CHAR and the CHAR_TO_LINE task itself. In this case, the first eighty calls to PUT_CHAR will simply copy their data into the character queue established by the *entry* declaration and proceed, even if the CHAR_TO_LINE task is unable to complete the rendezvous for the PUT_CHAR *entry* (presumably because it is waiting for a call to GET_LINE). Thereafter, additional calls to PUT_CHAR will block and be suspended until characters are taken from the queue by the CHAR_TO_LINE task.

For the most part, the implementation of this extension to the rendezvous mechanism is completely straightforward. For the case of an *entry* which has only *in* parameters, the calling task performs one of two actions when making an *entry* call. If the queue is not full, the input parameters are copied into the pre-allocated data area and added to the end of the queue; if the queue is full, the task activation record is queued for that *entry* in exactly the same manner as that used in the complete rendezvous approach. The server task, upon reaching an *accept* statement, looks to see if the queue is empty. If so, the server task is dismissed and waits for an *entry* call; if there are entries in the queue, the data items from the first *entry* are copied into the server task. As part of the same operation, the parameters from the first task (if any) in the associated queue of sending tasks must be appended to the end of the data queue, at which point the sending task is free to proceed.

A similar mechanism can be used to handle the case of entries which operate in the opposite direction and have only *out* parameters. In this case, receiving tasks are suspended when the data queue is empty and the server must wait when the data queue is full.

This approach makes considerable sense if one argues that many applications require efficient message passing structures and that those structures should be incorporated into the language in a manner consistent with the existing mechanism for synchronization. One important observation about this approach is that the queue size information may be interpreted in the same fashion as a pragma statement which the translator is free to ignore. If some translator chooses to implement all *entry* calls using the complete rendezvous scheme, this will only affect the efficiency of the resultant program rather than the semantics.

4.1.4. *Communication through low-level facilities*

One further alternative to be considered is to provide low-level facilities for mutual exclusion which would allow programmers to implement other message passing disciplines. While we do not feel that low-level facilities are required for an efficient solution to interprocess communication, we believe that there are other independent reasons which argue for the introduction of such facilities. If these are provided, it may be unnecessary for the language to supply any additional mechanisms for

communication since it will be possible for the users to create additional structures to achieve the necessary level of efficiency.

4.2. Low-level synchronization facilities in Ada

A related problem which limits the potential efficiency of Ada arises from the lack of low-level facilities for protecting shared data against concurrent access. In Ada, the only mechanism available for providing mutual exclusion is the *entry* call. Although it is certainly true that this model is appropriate to a variety of task structures which arise in practical applications, there are limitations in the structure which will make it difficult to use Ada in those environments in which efficiency is of considerable importance.

4.2.1. *Synchronization and efficiency*

As noted in the previous section, the rendezvous mechanism requires two scheduling events for each execution of a critical region. While this cost may be reduced considerably through the use of alternative implementation strategies, even in the best of circumstances, there will be some overhead cost involved in context switching between the two tasks.

The actual impact of the rendezvous overhead depends on the frequency of access to shared data and on the size of the critical regions. If access to shared data structures is relatively infrequent, the scheduling overhead required to make these accesses will have a minor overall effect. Similarly, if the size of the critical regions is large (in terms of the amount of computation required) in comparison to the rendezvous cost, overall system performance is relatively insensitive to this delay.

On the other hand, consider the extreme case of an application in which access to shared data is frequent (such as of the order of 10 per cent of the instructions executed not counting those required for parallel control) and the size of a typical critical region is very short (perhaps as little as one or two instructions). In this case, system throughput is largely determined by the efficiency of the mutual exclusion mechanism. On most systems, it is possible to design interlock mechanisms based on busy waiting (often referred to as 'spin locks') which require very few instructions to implement. If such a mechanism is used, it is reasonable to expect that a typical cycle from one critical region to the next might require no more than twenty or thirty instructions, assuming that lock contention does not have a significant effect. If scheduling interactions are required to ensure mutual exclusion, the path through a critical region would be significantly more costly and would typically require more than 200 instructions, thereby reducing the overall efficiency by an order of magnitude.

While the severity of the problem is exaggerated by the example above, the ratio of synchronization time to time spent in critical regions is an important factor in many applications. Furthermore, the choice between spin locks and scheduler-based synchronization mechanisms does have a significant impact on synchronization time. In the Hydra system, for example, spin locks are two orders of magnitude faster than the fastest synchronization primitive involving the scheduler (Reference 4). Since spin locks can be implemented using between three and ten instructions on most machines, this factor of 100 is likely to be representative of the relative cost for a wide range of systems.

The effect of this differential in the efficiency of the various synchronization primitives is that different applications may require different mechanisms according to the size of the critical regions involved. After studying the performance of a parallel root-finding application on C.mmp, Oleinick and Fuller[4] conclude that each of the scheduling mechanisms supported by C.mmp or the Hydra operating system has an associated operating range. If the time between synchronization events is relatively short (in this case, less than about 15 millisceonds), spin locks are the only syncronization mechanism available which incurs a synchronization cost of less than 50 per cent. If the interval between synchronization events is longer, the more powerful primitives provided by the scheduler become less costly.

The existence of different operating ranges suggests that some flexibility must be available in the choice of scheduling primitives in order to allow the system to meet the requirements of a particular application. The lack of this flexibility in Ada implies that the language may not be appropriate to applications in which the expected time between synchronization events is small. In our experience, this is frequently the case in real-time applications and we feel strongly that the introduction of low-level synchronization primitives into the Ada languague is necessary to handle such applications with the required level of efficiency.

4.2.2. *Control-based* vs. *data-based synchronization*

In addition to the efficiency concerns discussed in the previous section, the rendezvous mechanism in Ada differs from many conventional primitives for synchronization in that mutual exclusion is a function solely of the task (or control structure) and is independent of the data structures in the application program. This property appears to have an effect on memory utilization if conventional program structuring is used.

Consider an application in which some relatively large number of entities may be manipulated by some moderately large number of actions (for concreteness in this example, assume that there are 100 entities and 10 actions) in such a way that mutual exclusion is required to prevent two actions from occurring simultaneously for the same entity. This type of situation occurs, for example, in the case of a terminal concentrator whose function is to connect some large number of terminals to a network of host computers. In designing software for such a system, it is convenient to represent each terminal as a distinct entity and to define a set of commands which trigger control functions when entered on that terminal.

In Ada, this situation would ordinarily be modelled through the use of a *task family* whose members corresponded to the individual terminal entities. The user commands correspond to entries in the body of the task, which would give rise to the following general structure:

```
task ENTITY (1 .. 100) is
   entry ACTION1;
   entry ACTION2;
   -- entry declarations for remaining actions --
   entry ACTION10;
end ENTITY;
```

```
task body ENTITY is
begin
  loop
    select
      accept ACTION1 do
        -- body of action 1 --
      end ACTION1;
    or accept ACTION2 do
        -- body of action 2 --
      end ACTION2;
    -- accept statements for remaining actions --
    or accept ACTION10 do
        -- body of action 10 --
      end ACTION10;
    end select;
  end loop;
end ENTITY;
```

In a more conventional approach in which low-level primitives are available for locking within data structures, the same structure would be implemented by including an interlock with each entity to prevent concurrent access to that entity by more than one action. The individual actions would be coded as procedures, for example:

```
-- INTERLOCK operations defined in Section 3 --

type ENTITY is access
  record
    ACCESS_LOCK : INTERLOCK := UNLOCKED;
    -- local state fields --
  end record;

procedure ACTION1 (ENT : in ENTITY)
begin
  LOCK (ENT . ACCESS_LOCK);
  -- body of action 1 --
  UNLOCK (ENT . ACCESS_LOCK);
end ACTION1;

-- ACTION2 through ACTION10 are similarly defined --
```

The flavour of the two models above is very similar, particularly from the external point of view. In order to perform ACTION3 on some entity k in the task-based Ada approach, one issues the call

```
ENTITY(k) . ACTION3;
```

while in the interlock model, one performs

```
ACTION3 (pointer to entity k);
```

The semantic properties are also similar since each call is protected against the concurrent execution of other actions for that entity and independent entities may be acted upon in parallel.

In the implementation of the two mechanisms, however, there is a considerable disparity in the storage requirements which arises from the fact that the interlock model views the entities (data) and the actions (procedures) as distinct units. In the task model, each entity in the task family has, as part of its structure, each of the associated entries, which has a multiplicative effect on the storage requirements for each entity. For example, in the interlock model, there are 100 data locks used to manage concurrency; in the task model, this function is managed by 1000 (i.e. 100×10) entries. Since each *entry* must included at least a queue pointer, this approach is clearly inefficient in terms of storage.

It is possible to design the task structure for a particular application so that this cost is eliminated. For example, in the code sequence below there are only 100 entries to perform the necessary actions.

```
type ACTION is (ACTION1, ACTION2, ..., ACTION10);

task ENTITY (1 .. 100) is
  entry PERFORM_ACTION (ACT : in ACTION);
end ENTITY;

task body ENTITY is
begin
  loop
    accept PERFORM_ACTION (ACT : in ACTION) do
      case ACT of
        when ACTION1 =>
          begin
            -- body of action 1 --
          end;
        when ACTION2 =>
          begin
            -- body of action 2 --
          end;
        -- when clauses for remaining actions --
        when ACTION10 =>
          begin
            -- body of action 10 --
          end;
      end case;
    end PERFORM_ACTION;
  end loop;
end ENTITY;
```

While the above solution has the desired effect of reducing the storage requirements, the overall structure has been sacrificed and the resultant program is considerably less natural than the earlier one. It may be possible for the translator to perform optimizations of this kind, but this seems like an exceptionally complex problem.

4.2.3. *Implementation of interlocks in Ada*

Although the rendezvous can provide the same functionality as programmer-accessible interlocks within the data structure, we feel that such interlocks are

necessary in order to allow multiprocessor systems to be implemented with the required level of efficiency. The two preceding sections demonstrate that the interlock model is more efficient than a straightforward implementation of the rendezvous scheme. Because efficiency is of critical importance in most multiprocessor environments, we are concerned that the failure of Ada to provide adequate facilities for low-level interlocks will considerably reduce the overall applicability of the language.

We also believe that low-level facilities for managing interlocks can be added to the language without any significant change in the underlying structure of Ada. One possibility is simply to incorporate the data type INTERLOCK and the procedures LOCK and UNLOCK as part of the Ada language. This solution is sufficiently general to satisfy the efficiency considerations and does so with a very minimal impact on the Ada language. A second alternative would be to define a new statement form, such as the *region* statement from Brinch Hansen,[6] which has the effect of ensuring mutual exclusion on a particular interlock throughout a sequence of statements. This alternative offers greater protection against improper use of interlocks at the cost of introducing new syntactic forms into the Ada language.

It is important to note that the implementation of semaphore operations through the use of a generic task (as suggested in the *Ada Reference Manual*) is not a sufficient solution to the mutual exclusion problem, even if these primitives are implemented using special hardware support. There are two problems associated with the P and V operations as defined in Ada. First, tasks (including these generic tasks) are not part of the data environment. One of the principal uses of an interlock in conventional systems is to protect some structure from concurrent access. In Ada, there is no convenient way to associate a semaphore with a specific data object. The best achievable solution is to use integer indices within the object to select the appropriate member of a semaphore family in a relatively cumbersome and obscure way.

The second problem stems from the FIFO semantics of the rendezvous mechanism in Ada. Although the *Ada Reference Manual* (page 9–11) notes that the fact that semaphores are 'predefined authorizes an implementation to recognize them and implement them making optimal use of the facilities provided by the machine or the underlying system', it is still impossible to achieve the efficiency of spin locks in this way without violating the FIFO semantics of the Ada rendezvous.

4.3. Entries and the name problem

Another major problem in Ada stems from the manner in which processes are named. In Ada, tasks which perform some particular set of operations for separate internal data structures or devices are grouped together to form array-structured task families. In order to refer to a specific incarnation of a task, we must specify both the name of the task and the index of the specific process. Furthermore, since tasks in Ada are not data objects, we must supply the name field explicitly in the source code. This treatment of processes has several deficiencies when compared to other structures which allow a more flexible naming scheme.

4.3.1. *Limitations of array functionality*

One concern that arises from the naming convention is that the array structure imposes a relatively arbitrary task structure which may not fit the nature of the application. Array structured task families are appropriate only when the process structure which they represent has a topology which behaves like an array. Other structures (particularly those which involve linked lists or other pointer-based structures) are cumbersome to implement in terms of a pre-supplied array structure.

This problem is similar to the problem of defining linked structures in Fortran or a similar language in which arrays are the primary compound structure.

As an example, let us again consider the case of the terminal concentrator example presented above. In this application, there are a large number of terminals of which only a relatively small fraction are likely to be connected at any given time. The activity for each terminal is monitored by a member of a task family which is assigned to that terminal as long as it is connected to the system. We assume that the total number of terminal tasks is constant (which allows them to be statically allocated) and that the association of terminals and tasks will change as terminals are connected and disconnected from the system. Ordinarily, there will be more terminal tasks than connected terminals at any particular time; these tasks remain idle until they are associated with a newly connected terminal.

The natural structure in which to store the idle terminal tasks is a linked free list. When a terminal is connected to the system, it is assigned to the first free task which is currently at the head of the list. When a terminal is disconnected, its associated process becomes idle and is linked onto the free list. These operations are natural in a structure which permits pointer operations; when faced with an array structure, one is faced with the choice of (1) searching for free entries, (2) dynamically compactifying the task table so that the active tasks are contiguous or (3) simulating the free list mechanism through the use of auxiliary arrays. These alternatives represent possible implementation strategies, but it is our contention that Ada prevents the most natural solution.

4.3.2. *The return address problem*

A potentially more serious problem posed by the naming convention is the 'return address problem' which is briefly considered in the *Ada Rationale* (page 11–40). The concern here is that a server task has no way to reply to the calling task which requests service unless the identity of the calling task is known at translation time. The problem is not one of authenticating a particular caller but rather one of identifying the calling task in some subsequent *entry* call.

Consider the case of a task whose function is to encrypt a message supplied by a caller and to return the encrypted message. In Ada, the canonical description for this type of server is illustrated below:

```
task ENCRYPTION_SERVER is
  PACKET_SIZE : constant INTEGER : = 256;
  type PACKET is array (1 .. PACKET_SIZE) of CHARACTER;
  entry SEND_NORMAL_MESSAGE (MSG : in PACKET);
  entry GET_ENCRYPTED_MESSAGE (MSG : out PACKET);
end ENCRYPTION_SERVER;

task body ENCRYPTION_SERVER is
  BUF : PACKET;
begin
  loop
    accept SEND_NORMAL_MESSAGE (MSG : in PACKET) do
      BUF : = MSG;
    end SEND_NORMAL_MESSAGE;

    -- code to encrypt data in BUF --
```

```
      accept GET_ENCRYPTED_MESSAGE (MSG : out PACKET) do
        MSG : = BUF;
      end GET_ENCRYPTED_MESSAGE;
    end loop;
  end ENCRYPTION_SERVER;
```

While the code above performs the encryption function in a straightforward way and allows arbitrary tasks to call the two entries, it is not optimal in all cases. One potential problem arises in *entry* definitions which make use of a *select* statement to allow the server task to wait for a number of possible events. Because the *select* statement can appear only within the body of the called task, there is an inherent asymmetry in the tasking structure. Suppose that the programmer using ENCRYPTION_SERVER wanted a task within the following logical structure:

```
task body CALLING_TASK is

  -- code which generates PLAINTEXT for encryption --

  SEND_NORMAL_MESSAGE (PLAINTEXT);
  loop
    exit when ENCRYPTION_DONE;
    -- do some other work --
  end loop;
  GET_ENCRYPTED_MESSAGE (CODED_MESSAGE);

  -- code to make use of CODED_MESSAGE --

end CALLING_TASK;
```

While it is not possible to code the calling task in this way directly (because there is no way to transmit the ENCRYPTION_DONE), this type of operation can be achieved if the roles of *entry* call and *accept* statement are reversed for the GET_ENCRYPTED_MESSAGE *entry*.

```
task ENCRYPTION_SERVER is
  PACKET_SIZE : constant INTEGER : = 256;
  type PACKET is array (1 .. PACKET_SIZE) of CHARACTER;
  entry SEND_NORMAL_MESSAGE (MSG : in PACKET);
end ENCRYPTION_SERVER;

task body ENCRYPTION_SERVER is
  BUF : PACKET;
begin
  loop
    accept SEND_NORMAL_MESSAGE (MSG : in PACKET) do
      BUF : = MSG;
    end SEND_NORMAL_MESSAGE;

    -- code to encrypt data in BUF --

    GOT_ENCRYPTED_MESSAGE (BUF);
  end loop;
end ENCRYPTION_SERVER;
```

```
task body CALLING_TASK is

  -- code which generates PLAINTEXT for encryption --

  SEND_NORMAL_MESSAGE (PLAINTEXT);
  loop
    select
      accept GOT_ENCRYPTED_MESSAGE (MSG : in PACKET) do
        CODED_MESSAGE : = MSG;
      end GOT_ENCRYPTED_MESSAGE;
      else
        -- do some other work --
    end select;
  end loop;

  -- code to make use of CODED_MESSAGE --

end CALLING_TASK;
```

Unfortunately, this organization is only effective if there is a single calling task or a single family of callers. In the case that the calling task is a member of a task family, the caller can pass the index of a member as an additional argument to SEND_NORMAL_MESSAGE and then use this index in the subsequent GOT_ENCRYPTED_MESSAGE call, as in

```
CALLING_TASK(TASK_INDEX).GOT_ENCRYPTED_MESSAGE (BUF);
```

It is impossible to write ENCRYPTION_SERVER as a general utility package which is available for use with any task that calls SEND_NORMAL_MESSAGE and defines an entry GOT_ENCRYPTED_MESSAGE for the reply. Because it is impossible to pass the identity of the calling task to ENCRYPTION_SERVER, there is no way for the server task to return the message to the appropriate caller. This restriction seems to preclude the development of task libraries comparable to subroutine libraries in a well-organized environment for software development.

4.3.3. *Tasks as data objects*

The obvious solution to both the array topology problem and the return address problem is to consider individual activations of tasks to be data objects which can be incorporated into arbitrary structures or passed as parameters to server tasks. This issue is briefly discussed in the *Ada Rationale* (page 11–39) and the notion of anonymous activation variables from the language Tartan is introduced. Such a mechanism could be incorporated into Ada if it were possible to overcome the additional problems associated with task variables. For example, assume that all activations of tasks are data objects of the type ACTIVATION_NAME and that each task implicitly defines the variable MY_NAME to be an identification of that activation.

The discussion of activation variables in the *Ada Rationale* correctly observes that the introduction of untyped task variables raises questions of strong typing similar to

those found with procedure parameters in languages such as ALGOL-60. For example, even though the task definition

```
task body GENERAL_SERVER is
  DATA           : PACKET;
  RETURN_ADDRESS : ACTIVATION_NAME;
begin
  accept SERVER_REQUEST (T : in ACTIVATION_NAME,
                         INPUT : in PACKET) do
    DATA : = INPUT;
    RETURN_ADDRESS : = T;
  end SERVER_REQUEST;

  -- perform appropriate manipulation on DATA --

  RETURN_ADDRESS'REPLY(DATA);
end GENERAL_SERVER;
```

solves the return address problem, the use of an untyped process variable T is dangerous because there is no guarantee that the process referred to by T has a REPLY *entry* or that its parameter structure is compatible.

This problem, however, may be solved by eliminating the untyped activation variables in favour of a strongly typed system of specific *entry* variables. For example, assume that the reserved word *entry* is usable as a type generating function in a similar fashion as *array*. It is then possible to declare a return address with no type ambiguity as illustrated below:

```
task body GENERAL_SERVER is
  DATA           : PACKET;
  RETURN_ADDRESS : entry (in PACKET);
begin
  accept SERVER_REQUEST (T : in entry (in PACKET),
                         INPUT : in PACKET) do
    DATA : = INPUT;
    RETURN_ADDRESS : = T;
  end SERVER_REQUEST;

  -- perform appropriate manipulation on DATA --

  RETURN_ADDRESS(DATA);
end GENERAL_SERVER;
```

In this case, the caller would issue the *entry* call

```
SERVER_REQUEST (MY_NAME'REPLY, INPUT_DATA);
```

thereby giving the complete (and unambiguous) address of the return *entry*.

There are other possible approaches to this problem; we have suggested the above scheme in order to demonstrate that strong typing considerations alone are not a sufficient justification for disallowing references to process activations within the data structure. We believe that the ability to code a general server with the ability to

correctly address a reply is of major importance to the design of a rationally structured parallel control facility and that some mechanism for performing this function should be determined and incorporated into the Ada language.

4.4. Flexibility in the scheduling discipline

One additional area of concern that has developed during our study of Ada is the question of whether the scheduling discipline provided by the language is sufficiently general to support applications with important timing constraints. In particular, we are concerned that Ada does not provide adequate control over the scheduling strategy and that the scheduling algorithm is likely to encounter a number of problems associated with 'cooperative scheduling'.

To illustrate this problem, imagine that Ada is chosen as the implementation language for the design and development of a timesharing system for a multiprocessor. It is convenient in such a system to represent the individual user processes as independent tasks in the timesharing structure. In order to achieve fairness, timesharing systems typically limit the run-time allowed to a process to some maximum unit of time. If this time period (or quantum) is exceeded, the process is forcibly descheduled to allow other processes to run. The performance of the typical timesharing system is quite sensitive to the size and dynamic behaviour of this quantum limit and it is important to be able to adjust this mechanism to conform to the loading demands.

In Ada, there is no apparent way to specify a run-time limit for a task nor is it possible for one task to control the scheduling or descheduling of another. Without this flexibility, it appears that there are only two possible schemes to provide fairness in a timesharing scheduler:

1. Depend on the Ada scheduling discipline for all scheduling and descheduling operations and ensure that the built-in mechanism provides all of the desired flexibility, presumably expressed in the form of pargma declarations to the compiler.
2. Design a scheduler which operates 'cooperatively' in the sense that the tasks themselves participate in the scheduling decisions. In this case, each task would be required to periodically check its accumulated run-time and dismiss itself through the use of a delay statement.

Obviously, each of the approaches outlined above is totally unacceptable for a timesharing application. The first either requires the system designer to change the structure of the implementation language or forces the system to make use of a built-in scheduling discipline which may be hopelessly inadequate to perform the more complex scheduling operations required of a timesharing system.

The second approach is equally unworkable in that it requires the compiler to perform complex path analysis and assemble code to poll the scheduler at acceptably frequent intervals. The problems that arise in this type of scheduling are so severe that this alternative tends to be rejected out of hand. In his assessment of the process scheduling facility in Ada,[5] Paul Hilfinger writes:

> It seems that the tasks being scheduled must be written to be aware of the fact that they are being scheduled, and to do appropriate sends or procedure calls at intervals. This is a violation of abstraction; no reasonable operating system in existence requires that its processes cooperate to be scheduled.

There are several potential approaches to this problem which affect the structure of the language to varying degrees. Perhaps the most straightforward mechanism is to allow one task to forcibly deschedule another task. This would provide a monitoring task with at least some primitive ability to control the scheduling discipline. This could be implemented through the addition of a new primitive such as

```
deschedule T;
```

or as an extension of the priority mechanism. If one task were allowed to alter the priority of another and changes in priority were implemented so as to force a scheduler transition, one might begin to have an acceptable facility for scheduling control.

5. CONCLUSIONS

In this paper, we have argued that multiprocessor systems are frequently used for real-time applications in which run-time efficiency requirements are of critical importance. For this reason, we believe that the design of a high-level language system which is intended for use in real-time, multiprocessor-based applications must be sensitive to these requirements and must allow the programmer to write code which satisfies the efficiency constraints imposed by the application.

We believe that the Ada language, as currently designed, does not meet these needs for several reasons:

1. The use of a complete rendezvous system results in unnecessary scheduling delays. This problem is particularly severe in the relatively important case of message passing in that Ada requires the sender of a message to wait for the scheduler before it is allowed to proceed. This structure is considerably less efficient than message passing systems implemented with queues and imposes a relatively high cost on the use of an important communication discipline.
2. Ada does not provide sufficent flexibility in its process control structure to allow the programmer to choose the mechanism most closely suited to the requirements of the application. In particular, the fact that the mutual exclusion mechanism is associated with the control structure rather than the data structure leads to convoluted program structures or serious inefficiencies in the use of space.
3. The naming conventions used to indicate specific processes in Ada are not sufficiently general to allow the programmer to represent process structures which accurately reflect the underlying structure of the algorithm. Moreover, the fact that no general mechanism exists to allow one process to communicate its identity to other processes in the system severely limits the modularity of the task structure.
4. The language does not provide the user with sufficient control over the scheduling discipline.

We contend that the parallel processing facilities currently provided by Ada do not satisfy the requirements of real-time systems such as those typically chosen for implementation on a multiprocessor. On the other hand, we feel that good solutions do exist for most of the problems that we have identified here and that those solutions can be incorporated into Ada with relatively little change to the overall structure of the code. Based on our experience with multiprocessors and real-time systems, we feel

that the efficiency cost implied by the current Ada design severely limits the extent to which Ada is acceptable for real-time applications. We strongly urge that modifications such as those suggested in this paper be incorporated into Ada to increase its utility in this important area of application.

ACKNOWLEDGEMENTS

The research reported in this document was sponsored in part by the Defense Communications Engineering Center under Contract No. DCA100–78–C–0028 and by the National Science Foundation under Grant No. MCS–7908365.

REFERENCES

1. J. D. Ichbiah *et al.* 'Rationale for the design of the Ada programming language, *SIGPLAN Notices*, **14** (6), Part B, entire issue (1979).
2. U.S. Department of Defense, *STEELMAN—Requirements for High Order Computer Programming Languages*, Defense Advanced Research Projects Agency, Arlington, Va., June 1978.
3. C. A. R. Hoare, 'Communicating sequential processes', *Communications of ACM*, **21**, 666–677 (1978).
4. P. N. Oleinick and S. H. Fuller, 'The implementation and evaluation of a parallel algorithm on C.mmp', *Computer Science Department Report CMU-CS-78-125*, Carnegie-Mellon University, June 1978.
5. D. A. Lamb (ed), *Commentary on the RED and GREEN Candidates for the Ada Language*, Computer Science Department, Carnegie-Mellon University, April 1979.
6. P. Brinch-Hansen, *Operating System Principles*, Prentice-Hall, Englewood Cliffs, N.J., 1973.