Timing Issues in the Distributed Execution of Ada Programs

RICHARD A. VOLZ, SENIOR MEMBER, IEEE, AND TREVOR N. MUDGE, SENIOR MEMBER, IEEE

Abstract—**This paper examines, in the context of distributed execution, the meaning of Ada constructs involving time. In the process, unresolved questions of interpretation and problems with the implementation of a consistent notion of time across a network are uncovered. It is observed that there are two Ada mechanisms that can involve a distributed sense of time: the conditional entry call, and the timed entry call. It is shown that a recent interpretation by the Language Maintenance Committee resolves the questions for the conditional entry calls but results in an anomaly for timed entry calls. A detailed discussion of alternative implementations for the timed entry call is made, and it is argued that: 1) timed entry calls imply a common sense of time between the machines holding the calling and called tasks; and 2) the measurement of time for the expiration of the delay and the decision of whether or not to perform the rendezvous should be made on the machine holding the called task. The need to distinguish the unreadiness of the called task from timeouts caused by network failure is pointed out. Finally, techniques for realizing a single sense of time across the distributed system (at least to within an acceptable degree of uncertainty) are also discussed.**

Index Terms—**Ada, Ada conditional entry calls, Ada task timing, Ada time entry calls, distributed Ada, distributed languages, distributed program execution.**

I. Introduction

ONE of the principal purposes for which the Ada language was designed is the programming of embedded real-time systems [1], and, with increasing frequency, embedded real-time systems involve distributed computing. It is therefore necessary that Ada support the distributed execution of programs. In this paper, we explore one of the most important factors in achieving distributed execution of Ada programs: the management of time across a network of processors. In particular, we examine the meaning of Ada constructs involving time in the context of distributed execution, note that there are both unresolved questions of interpretation and problems with the implementation of a consistent notion of time across a network, and propose interpretations and timing mechanisms to resolve these problems. Other important issues involving the distribution of Ada across a network of processors are discussed in [2]–[4].

The Ada mechanisms involving time are the *delay* statement, the conditional entry call, the timed entry call, and the selective wait statement. The *delay* and selective wait statements are strictly local in their actions (i.e., their effects take place on a single processor), and thus are not of concern in this paper. Of course, the view of time and the underlying mechanisms for managing it are crucial. The management of time in the distributed environment begins with the management of time within a single processor. This is discussed in a separate paper in which we recommend that time within a processor be kept in a (locally) absolute sense and show a mechanism for accomplishing absolute timing which is simpler than the mechanisms now in use [5]. In this paper, we will argue that, among other things, this absolute sense of time should be extended to the multiple distributed processor situation as well, and that, indeed, such an absolute network sense of time is required by the reference manual [1]. In doing so, we will focus attention on conditional and timed entry calls. An absolute sense of time is assumed in several models for real-time systems (see, for example, [6]), and the advantages of using absolute time are also discussed in [7], where it is proposed for fault-tolerant distributed systems.

The definitions of conditional and timed entry calls are not entirely clear when examined in the distributed setting. The interpretations applied significantly effect the implementation. We will approach the problem by trying to make a strict interpretation of Ada as presented in the reference manual (RM), since the "no supersets, no subsets" philosophy is one of the major tenets of the language and the principal upon which the portability of Ada is based. Where there is possible ambiguity in the interpretation of the manual due to consideration of distributed execution, the various possibilities and their implications are discussed. We expect that, ultimately, the Ada Board and ISO Working Group 9 will have to examine these problems and issue interpretations of the RM to cover the distributed situation more completely. It is hoped that the discussions presented here will aid in the determination of logically consistent and implementable interpretations.

In the next section we review the conditional and timed entry call structures of Ada to place the rest of the paper in perspective. Section III then examines the issues in conditional entry calls, while Section IV does so for timed entry calls. In both sections interpretations are proposed. Section V follows by addressing the question of maintaining the network sense of time required for a reasonable interpretation of time entry calls. Concluding remarks are presented in Section VI.

II. Overview of Conditional and Timed Entry Calls

The conditional entry call is used to determine whether or not the called task is ready to accept an entry call and, if it is,

Manuscript received August 28, 1986; revised December 3, 1987. This work was supported in part by General Dynamics under Contract DEY-605028, General Motors Corporation under Contract GM/AES (1986–87) and NASA under Contract NAG 20359.

The authors are with the Robotics Research Laboratory, College of Engineering, University of Michigan, Ann Arbor, MI 48109.

IEEE Log Number 8613057

to make the call. To illustrate, suppose that a robot and an automatic guided vehicle (AGV) are engaged in a cooperative manufacturing task in which the robot unloads two different kinds of parts from a pair of machine tools, placing them in a temporary storage area, and when the AGV is ready, loads parts onto it. The AGV alternately is loaded with parts by the robot and transfers them to a longer term storage area where it is unloaded and then returns to the robot for another load. We assume that the temporary storage area always contains enough parts to fill the AGV. An abstraction of the relevant parts of the robot and AGV tasks might look like the following.

Example 1

Abstraction of Robot Task:

```
.
.
loop
  select—This begins a conditional entry call.
    AGV.READY(KIND);--This is the actual call.
    --Load a part of type KIND on the AGV from
    --temporary storage.
  else
    null;
  end select;
  --Unload a part from a machine tool
  --and place it in temporary storage.
end loop;
.
.
```

Abstraction of AGV Task:

```
task AGV is
  entry READY(K: out TYPE_OF_PART);
end AGV;
task body AGV is
.
.
while AGV_NOT_FULL loop
  accept READY(K: out TYPE_OF_PART) do;
    K: = LOCAL_KIND_NEEDED;
  end READY;
end loop;
.
.
```

Upon reaching the **select**, the robot task would check to see if the AGV task has reached the **accept** READY statement. If it has, it will rendezvous with AGV at that point. The rendezvous consists of executing the code between the "**accept** READY" and the "**end** READY." In this case, the rendezvous simply amounts to invoking a function LOCAL_KIND_NEEDED to determine the type of the next part to be loaded, and then assigning the result to KIND, a variable of the robot task. (KIND is the actual parameter for K in the rendezvous executed by the AGV task.) The value of KIND tells the robot what kind of part to load next. After completion of the rendezvous, the robot will proceed to load the AGV with the appropriate part. Following this, the robot will unload the next part from the machine it is tending. In the case where the AGV has not reached the READY entry point at the time that the conditional entry point is made, then the robot task will start immediately to unload the next part from the machine.

Whenever the AGV is ready to be loaded with parts by the robot, it will reach the loop shown above. If it reaches the the **accept** statement before the robot makes the call, it will simply wait at that **accept** statement until the robot task makes the call to READY. After each rendezvous, which initiates loading another part, the function AGV_NOT_FULL is called to check if the AGV has room for more parts (it returns a Boolean value).

We use the same example to illustrate timed entry calls. Consider the code abstraction shown below.

Example 2

Abstraction of Robot Task:

```
loop
  select--This begins a timed entry call.
    AGV.READY(KIND);--This is the actual call.
    --Load a part of type KIND on the AGV from
    temporary storage.
  or
    delay 1*SECOND;--The time limit for accepting
    the call.
  end select;
  --Unload a part from a machine tool
  --and place it in temporary storage.
end loop;
```

The abstraction of the AGV task is the same in this case as for the conditional entry call. The operation in this case is similar, except that the robot task will now wait one second after attempting the call to AGV.READY before taking the alternative of unloading a part from the machine it is tending.

Note that the segment of code which is executed during the rendezvous is written as part of the called task. Normally, this will mean that this segment of code will be located on the processor holding the called task. However, Haberman and Nassi [8] have shown that, in some cases, this code may be executed in the context of the calling program for the purposes of reducing execution time. We will consider the implications of both locations in the following discussion.

III. Conditional Entry Calls

First, we examine an ambiguity in the interpretation of conditional entry calls across a network of processors. The RM, in Paragraph 1 Section 9.7.2, states that "A conditional entry call issues an entry call that is then canceled if a rendezvous is not immediately possible." There is a possible difficulty in the word "immediate." At least one group [9] has interpreted the work "immediate" in a temporal sense and used this to disallow conditional entry calls when such calls are placed across the network since network delays would prevent the "immediate" determination of whether or not the call

could be accepted. This would mean that in the example above, the conditional entry call from the robot task to the AGV task would always fail and the code sequence shown could not be used to cause the AGV to be loaded. One would be forced to use the timed entry call.

However, the RM also presents a nontemporal interpretation of the word "immediate". In Paragraph 4 of the same section it restates the conditions for cancellation of the call: "The entry call is canceled if the execution of the called task has not reached a point where it is ready to accept the call...." There is nothing that inherently involves time in this interpretation. This statement expresses the action of the conditional entry call only in terms of the readiness of the called task to receive the call. This is appropriate, and this interpretation shall be used throughout the remainder of this paper. If a sense of time is required, timed entry calls should be used.

In a related matter, the RM, in Paragraph 4 Section 9.7.3, states that timed entry calls with zero or negative delays are to be treated as conditional entry calls. Under the condition that the called task is ready to accept a call, an inconsistency may arise with respect to whether the rendezvous should be completed or canceled. Due to delays in network transmission, there will be a set of small delays for which the rendezvous fails, while for delay values either above or below those in the set, the rendezvous would succeed. This situation is illustrated in Fig. 1 where we have shown one of the possible protocols for managing the remote timed entry call (others are discussed in Section IV-D). If a call is initiated on processor A at time t_1, it is not received on processor B until time $t_1 + d_n$. If the specified delay is less than d_n, the delay will have expired and the request will be denied. If the specified delay limit is greater than the network delay time, the call will succeed. However, the call succeeds for zero and negative delay since it is then treated as a conditional entry call. A more consistent statement would result if the RM did not contain the phrase about treating the case with zero or negative delay as conditional entry calls. Nevertheless, the RM does state quite clearly that the situation is as shown in Fig. 1.

These questions of interpretation of conditional entry calls have been considered by the Language Maintenance Committee of ISO Working Group 9 and the Ada Board of AJPO. In Ada interpretation number AI 276 the committee has stated roughly the interpretations expressed here.

IV. Timed Entry Calls

Timed entry calls are not as easily handled as conditional ones; the anomaly of Fig. 1 is only part of the problem. They raise a number of issues, not only about the interpretation of the timed entry call itself, but about the management of time in a distributed environment as well. The timed entry call is the one place in the RM where an upper bound is placed on the time duration for some action to occur. This is both necessary and the source of interpretation and implementation difficulties in a distributed environment. We interpret this upper bound in a strict global absolute sense. That is, the stated action must be accomplished within the required time in spite of network time delays or failures, or the alternative action must be taken.

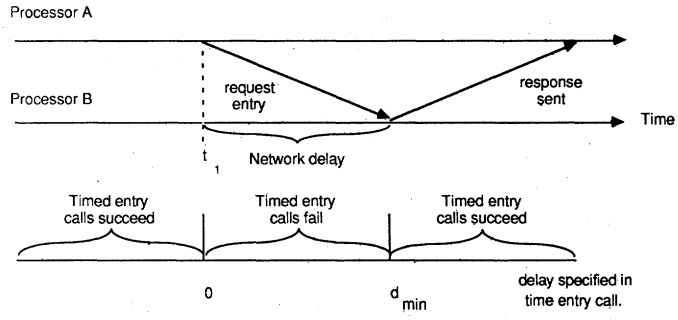


Fig. 1. The timed entry call anomaly.

There is, of course, a trivial implementation of the timed entry call. One could say that since one cannot, in general, exactly maintain a network sense of time, rendezvous for timed entry calls never take place and the calling unit always executes the alternative sequence of code. However, this is unnecessarily restrictive, timed entry calls are a valuable part of the language, and it is possible, and thus important, to find consistent interpretations and implementations for them, even in the distributed environment.

A. Review of Ada Timed Entry Calls

To begin a study of timed entry calls, we review the relevant statements from the RM. In Paragraph 4 of Section 9.7.3, the RM says both that:

1) "If a rendezvous can be started within the specified duration..., it is performed...."

2) "....the entry call is canceled when the specified delay has expired, ..."

Statement 1) refers to an action performed on the processor containing the called task, while Statement 2) refers to an action performed on the processor containing the calling task. Implicit in these statements is thus the notion that there is a common sense of time between the calling and called processors. This common sense of time must be maintained in the face of network delays, clocks on individual machines that are not precisely synchronized, as well as failures in the system. In general, of course, this cannot be done exactly. One must develop interpretations that take into account disparities in the clock measurements made at different parts of the system. We will, however, initially develop our interpretations assuming a perfect common network sense of time, i.e., if read at the same time, clocks on all processors would yield the same value. We will also initially assume a constant network communication delay d_n on all interprocessor messages. Later we will relax these assumptions and extend our interpretations to handle variations in time that exist in practice.

The principal difficulty with interpretating these two statements in the distributed environment arises because information must be transmitted between the calling and called processors, and this transmission typically takes a significant amount of time. Because of this network transmission time, it is not possible to operate in a manner that satisfies both Statements 1) and 2) simultaneously.

Before discussing this problem in detail, we elaborate further on the implications of 1) and 2). Consider a timed entry

call from a task executing on processor A to an entry of a task located on processor B. The entry call is made at time t_1 and has a delay of d. Then the time $t_2 = t_1 + d$ is the time by which the called task must be able to accept the call. Taken literally, Statement 1) says that if by the time t_2 the called task has reached an appropriate accept statement the called task is made ready so that the rendezvous may take place. Similarly, 2) says that if by time t_2 the called task has not reached an appropriate accept statement the call is canceled and the calling task is made ready at the alternative sequence of statements following the **delay** part of the timed entry call. In neither case does the language require that the rendezvous or the alternative sequence of statements actually start, just that they be made ready within the stated time interval. Actual starting times will depend upon other tasks, and their priorities, that are also ready, and upon the scheduling mechanisms used.

This lack of rigid upper bounds on the actual start of actions ensuing from a timed entry call might be used as an excuse for relaxing the rigid bounds implied by 1) and 2) on the times at which the tasks are made ready. Indeed, we will show that this is necessary for one or the other of the two statements. In spite of this, however, we believe it is necessary to maintain rigid bounds where possible. An application may (particularly in the distributed situation) only have a single task on a processor, in which case the task would resume shortly after being scheduled, and an application might depend upon bounding this time. Furthermore, if the time bound were relaxed with respect to both statements, the timed entry call would have no meaning at all. We will show that under certain reasonable conditions the bound specified in 1) can be realized.

A slightly stronger interpretation, and one which is probably more difficult to implement, would result if 1) were interpreted to mean that the called task must actually start by the time t_2. This would add little, however, since the called task could always be preempted by a higher priority task. What might be useful would be to bound the completion time of a rendezvous. Although Stankovic [10], [11] discusses techniques that can guarantee ending times of tasks, these techniques require more information, e.g., a global view of tasks to be scheduled and their repetition rates, than are available in an Ada timed entry call statement; they thus cannot be automatically constructed from the data associated with the timed entry call.

B. Discussion of Problem

To illustrate the impossibility of simultaneously satisfying Statements 1) and 2), we describe one (of many) protocols which might be used in implementing timed entry calls.

Example 3: We consider the communication sequence shown in Fig. 2. At the time t_1 a timed entry call is encountered, and a message is sent from processor A to processor B indicating that the rendezvous is requested. This message is received by B at time t_a. It contains as a parameter the time $t_2 = t_1 + d$ by which the rendezvous is to be accepted. The measurement of the time t_2 is performed on processor B and the decision of whether or not to accept the call also made on processor B. Two cases are shown. For case

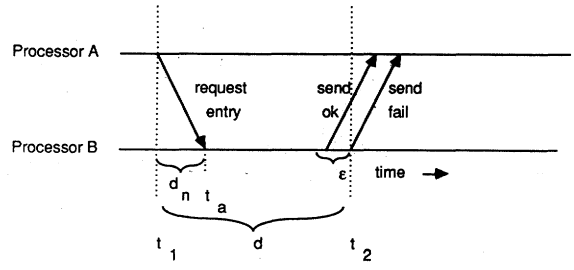


Fig. 2. Communication sequence for Example 3.

1, the called entry is able to accept the call at the time $t_2 - \epsilon$ and the rendezvous is accepted. For case 2, time t_2 is reached without the entry call being accepted and the timed entry call fails. In the case that the rendezvous is accepted, the called task is immediately made ready on processor B and will execute in accordance with task scheduling mechanisms in use on processor B. When the rendezvous is completed, a message is sent to processor A indicating the completion. Statement 1) is thus satisfied. Note that processor A cannot know whether or not the call was accepted until some time after t_2, and that this violates a strict interpretation of Statement 2). It is only possible to cancel the call some time *after* t_2, possibly as much as d_n after t_2. We will show later that there exists a different protocol which would allow 2) to be satisfied at the expense of 1).

One set of issues, then, is which of the Statements 1) or 2) is to be satisfied and how this is to be done. We will refer to these two choices as interpretations I1 and I2, respectively. There is, however, another aspect to the question which must also be considered at this point: the use to which timed entry calls are put. Until now, the discussion has been phrased in terms of determining the readiness of the called task to accept an entry call as this is the obvious interpretation from the RM. One might also consider using them as timeouts for detecting network or other system failures. The network might fail at any of several points in the communication sequence, or the processor on which the called task resides (or the device associated with the entry point) might fail. By basing the interpretation of timed entry calls on Statement 2), one might detect such failures through timeouts. The use of timed entry calls for this purpose impacts the possible protocols and interpretations of Statements 1) and 2) and thus will be considered here.

There are three possible things one might try to accomplish with timed entry calls:

- establish a bound on the time at which a rendezvous is scheduled to start (I1),
- establish a bound on the time at which the delay in awaiting a rendezvous expires (I2), and
- detect network or system failures.

These goals are not mutually compatible and we will explore the differences below.

Example 3 above assumed both a goal of achieving interpretation I1 and that the time measurements and the decision process were performed on processor B. A communication sequence was then selected to achieve I1. Actually, there are two possible interpretations, I1 and I2, and two

locations at which the time measurements and decisions could be made. There are thus four basic cases to consider, with variations on each as to the locations at which the rendezvous code could be located:

Case 1: Interpretation I1 and decision on called processor.

Case 2: Interpretation I2 and decision on called processor.

Case 3: Interpretation I2 and decision on calling processor.

Case 4: Interpretation I1 and decision on calling processor.

Before discussing these cases, however, we will return to the question of the use of timed entry calls and argue that they should not be used for device timeouts and system failure detection; instead we will argue that exceptions should be used. This discussion impacts the subsequent discussion on the interpretation of Statements 1) and 2).

C. Timeout Detection of System Failures

The use of timed entry calls for failure detection implies, first of all, a measurement of time on the calling processor since the failures being tested for could preclude receipt of a value of time measured at any other location. This either limits the protocol choices or requires the determination of the lapse of the time interval on both the calling and called processors. In either case, unfortunately, if timed entry calls are used to detect network, node or device failures (any of which we will call a system failure), there is a possible ambiguity in the interpretation of the expiration of the delay. One cannot know whether it means merely that the called task has not reached an appropriate **accept** or whether there has been a system failure.

As an illustration, consider an extension to Example 3 in which the expiration of the time delay is measured on both processors A and B, and that a network failure occurs at time $t_2 + \delta$, before the messages can reach processor A. The calling task will eventually time out, and have no way of knowing whether or not the called task was able to accept the call. This means that the alternative part of the timed entry call must be prepared to deal with an indeterminate situation.

To solve this problem, one must first recognize that there are two distinct types of conditions to be detected, the readiness of the called task and system failure detection. We believe that two distinct techniques are required. In particular, we believe that the timed entry call should be used for determining the readiness of the called task and that **exceptions** should be used for device timeouts.

The use of **exceptions** to handle device timeouts seems more natural than using timed entry calls since a network or other system failure is, in fact, an exception to normal operation and would seem to fit the role for which **exceptions** were intended. Also, the actions which must be taken to recover from a timeout can be more drastic than those required from a task being unready to accept a call. For example, in the illustration described above, the called task may have started the rendezvous when the calling task times out, requiring the recovery procedure to roll back the effect of the rendezvous.

In order to use exceptions in this way, an implementation could include a generic **package** TIMEOUT that provides an **exception** and associated data and operations. A data object of type DURATION would be needed for each instance of TIMEOUT, and procedures would be needed to set this value and initiate timing. Each task using TIMEOUT exceptions could instantiate an instance of this generic package to provide an actual exception and associated objects.

```
generic
package TIMEOUT is
  LATE_START: exception;
  procedure SET_START_LIMIT(DEL: DURATION);
end TIMEOUT;
```

Fig. 3.

The generic package TIMEOUT must essentially provide two kinds of things,

- functions which take an object of type DURATION as an argument and activate a timeout, and
- exceptions which are raised if the timeouts expire.

The details of the generic TIMEOUT package depend on the interpretation of Statements 1) and 2) and the protocol implementing them.

To illustrate, we will again extend Example 3. The **delay** associated with the timed entry call will be used, as illustrated previously, for the obvious purpose of checking the readiness of the called task. The generic package TIMEOUT will provide an additional timeout for failure detection. Fig. 3 shows the specifications of a generic package intended to be used with the protocol of Example 3. It provides an exception LATE_START, and the procedure SET_START_LIMIT which provides a link to the runtime system and defines the additional timeout. The effect of this timeout is not immediate, however. It is activated upon the next timed entry call. If an entry call acknowledge message is not received within the specified time limit after the beginning of the entry call, the exception LATE_START is raised.

With the use of TIMEOUT, the timed entry call of Example 3 would take the following form.

```
EX3_TIMEOUT is new TIMEOUT;
.
.
EX3_TIMEOUT.SET_START_LIMIT (2.0*
   NETWORK_DELAY);
select
  REMOTE.ENTRY(PARAMETERS);
or
  delay DELTA;
  --alternative sequence of statements if the called task
  --is not at an appropriate accept
end select;
.
.
exception
  when EX3_TIMEOUT.LATE_START = >
  --corrective action
.
.
```

The interpretation of the timed entry call would then be in accordance with Statement 1). The cancellation of a call due to expiration of DELTA occurs only when processor A receives

a message from processor B indicating that the rendezvous could not be accepted in time. If a message indicating success or failure is not received by the time $t_3 = t_2 + 2d_n$, it is assumed that there has been a system failure and LATE_START is raised. Note that with the addition of TIMEOUT, the alternative sequence of the timed entry call always refers to the failure of the called task to reach an appropriate **accept** within the desired time; it never refers to a system failure. System failures are always handled by exceptions, which is in line with the intent of exceptions. If it were possible to actually bound the network transmission time by d_n, then the **exception** LATE_START would always mean system failure and we would have orthogonality of the two constructs (timed entry call and timeout detection of system failures). While such a bound will not exist in all circumstances, in practice it may exist in a very large percentage of situations.

This example does not provide for any error checking on the ending time of a rendezvous, or system failure during the message exchange at the conclusion of the rendezvous, but then, neither does Ada. One could handle the possibility of detecting system failures during a rendezvous by including additional exceptions and procedures in the generic package TIMEOUT. These will not be discussed here, but deferred to discussions of individual protocol and interpretation options.

If one did not use the TIMEOUT package, the protocol of Example 3 would have to be changed or the system could hang forever on a system failure. One possibility would be to perform timing on both processors and make an explicit check of the system when the calling processor detects the elapse of the **delay** specified. This is similar to the protocol suggested in [12] in which the calling processor tries to withdraw the request at the expiration of the **delay**. However, the approach of [12] has two negative features. First, it removes some of the orthogonality of language features. Second, it requires extra overhead in the usual situation in which the expiration of the time delay simply means that the called task has not reached an appropriate accept.

We will thus consider timed entry calls to be used for testing the readiness of the called task and not for detecting system failures.

D. Alternative Interpretations of Statements 1) and 2)

The following sections will address five basic protocol types and interpretations of Statements 1) and 2) for dealing with timed entry calls. These correspond to the four cases listed above and a variation of the location of the rendezvous code. As appropriate, additional TIMEOUT procedures for failure detection will be discussed.

Case 1) Interpretation I1 and Decision on Called Processor: Consider first taking the called processor as the point of decision and reference for time measurements. This is essentially the situation illustrated in Example 3 above. The principal question with this protocol is the interpretation of Statement 2) which calls for cancellation of the entry call when the delay has expired, i.e., at time t_2. As illustrated in the example above, if one makes the decision about accepting a timed entry call on processor B at time t_2, then it is not possible for processor A to make a decision about canceling the call "when the delay has expired," i.e., also at time t_2. However, if one makes a liberal interpretation of Statement 2), then canceling the entry call only means taking the alternative sequence at some time after t_2, and taking the alternative sequence (if present) at time $t_1 + d + d_n$ on processor A would be consistent with 2). The decision of whether or not to cancel the call is then not directly dependent upon time, but depends only upon receipt of the appropriate message from processor B.

With this interpretation and communication sequence the timed entry call is written assuming that no relevant network or system failures occur. A secondary means, such as the TIMEOUT.LATE_START exception described above is required, and provides an adequate means, for detecting failures during the initiation of the rendezvous.

Detecting failures during the rendezvous or the completion message transmission is a bit more complex. There are two obvious possibilities. First, if the user can be expected to place an upper bound, say d_R, on the time to perform the rendezvous (including any delays accruing from interrupts of higher priority tasks) then a second procedure SET_RENDEZVOUS_LIMIT(..) could be added to the generic package TIMEOUT together with a second exception LATE_FINISH, which has the effect of raising the exception if a completion message is not received at the calling processor within the duration specified in the argument. The obvious difficulty with this approach is the existence of the bound d_R; in general, one will not be able to place such a bound on the system. Without an upper bound, the occurrence of the exception could, in some cases, represent delays introduced through response to higher priority tasks rather than a system failure. The exception handler would then have to perform explicit checks to determine the actual situation.

Second, a double phase completion protocol, as shown in Fig. 4, could be used. In this case, processor A must acknowledge the receipt of the completion message from processor B. Processor B performs the timeout check. It could either use the same duration specified in SET_START_LIMIT or use a separate procedure to specify the limit. A system failure would then be detected on processor B, which would then raise the exception LATE_FINISH if the acknowledge were not received in time. This case, however, only checks the system during the completion message exchange; it does not provide any detection capabilities for failures during execution of the rendezvous. It is likely that if this option is chosen an implementation would provide, as an implicit parameter, information about the lineage of the task on processor A so that processor B could report the failure to the appropriate parent task.

We note that this protocol is similar, in some respects, to the protocol suggested in [12], in which decisions are made on both processors. In that proposal no message is sent from B to A to indicate acceptance of the call. Instead, if the delay expires on the calling processor A, a message is sent to B at time t_2 asking to withdraw the rendezvous request. If the rendezvous was actually started within the requested time, the withdrawal request is denied and the rendezvous proceeds. The calling processor cannot know until two message times

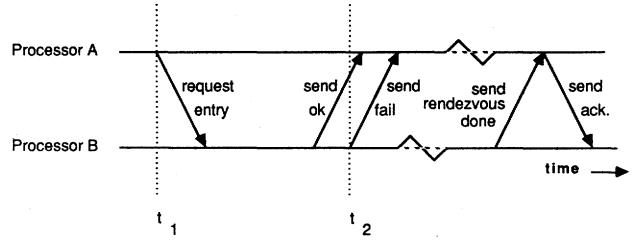


Fig. 4. Communication sequence indicating end of rendezvous.

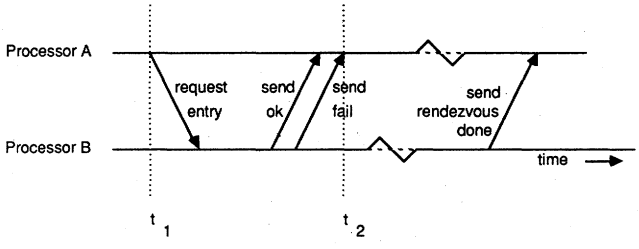


Fig. 5. Communication sequence for Case 2.

after t_2 whether or not the rendezvous is proceeding. Assuming no network failures, the effect is similar to the protocol described here except that a larger delay (two network message times) can occur before the canceling of a timed entry call due to the unreadiness of the called task.

Case 2) Interpretation I2 and Decision on Called Processor: In this case, processor B makes the decision and must notify processor A by time t_2 whether or not the call can be accepted with the given time interval, as shown in Fig. 5. In order to do this, processor B must be able to bound the network delay and make the decision prior to time t_2. Thus, the interpretation of Statement 1) must be relaxed and the decision point moved up in time. This is analogous to the relaxation of Statement 2) which was made in Case 1.

The difficulty in this case is the need to bound the network delay, d_n. When one considers the possibility of transmission errors and retransmissions, this is not strictly possible. Also, for many networks, the message transmission time, even for successful transmissions, cannot be bounded. Thus, this case will not be considered further.

Case 3) Interpretation I2 and Decision on Calling Processor: A communication sequence to achieve this combination is shown in Fig. 6. The shaded arrow from A to B at the beginning of the sequence is an optional message in the sequence. The solid arrow from B to A is the upper bound on the time at which the message could be sent while the shaded arrow from B to A indicates that it could be sent at any earlier time. The essential point is that processor B notifies processor A when it is ready to accept an entry call. If processor A has received a ready message from B by the time t_2 the call is accepted; if not, the call is canceled at time t_2. Once processor A makes the decision, a message is sent to processor B indicating whether or not the rendezvous is to be performed. In this case, the time of making the decision in the task containing the rendezvous code segment ready is relaxed. The notification that B is ready to accept a call may either be in response to a request from processor A (shaded arrow at the beginning of the sequence) or a broadcast to all that it is ready.

This case is essentially the dual of Case 1 in the sense that

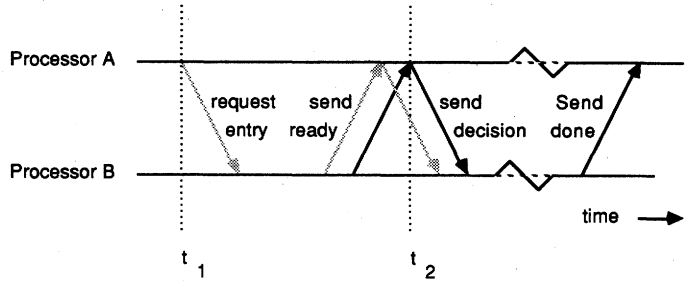


Fig. 6. Communications sequence for Case 3.

the roles of A and B in timing and decision making are reversed. However, contrary to Case 1, if the call is canceled, one cannot know if it is due to the unreadiness of the called task or a system failure. If it is important to make this distinction, the alternative code sequence must explicitly check the system status. This impairs the orthogonality of the construct to other mechanisms for handling errors. Also, the task executing the rendezvous is not made ready until after a message is received from A indicating that the rendezvous is to be performed. A network failure could occur during the transmission of this message and the system would hang.

In comparing Case 1 to Case 3, we make several observations. Both achieve an upper bound on making either (but not both) the rendezvous code or the alternative code sequence ready under the conditions that the given code section is selected. Case 1 achieves the bound on the rendezvous code, while Case 3 achieves it on the alternative sequence. Second, Case 1 appears to be somewhat more amenable to achieving orthogonality of the language than Case 3. Also, since accomplishing the rendezvous within a given time interval would seem to be the intent of the timed entry call, placing the bound on the rendezvous code would seem more natural than placing it on the alternative sequence. For these reasons, we prefer interpretation Case 1.

Case 4) Interpretation I1 and Decision on Calling Processor: Fig. 7 illustrates a message sequence for accomplishing this case. As with the previous case, the called task must notify the calling task that it is ready to accept an entry call. It may do so either by responding to a entry request or in a broadcast mode. The difference between this mode and the previous case is that the calling processor must anticipate the network time delay and make the decision far enough ahead of time to allow the message containing the decision to reach the called task by time t_2. However, since the time reference is on the called processor, the decision time must be advanced further than in Case 3 so that the decision can reach the called processor by time t_2. The amount of time by which statement (1) must be relaxed is thus greater than in Case 3. Further, this case depends upon the bounding of the network time, which is an undesirable feature. Thus, this case does not appear to have any advantages, and will not be considered further.

Case 5) Same as Case 3 with Rendezvous Code on Calling Processor: It has been suggested that for purposes of optimization the code associated with the rendezvous could be placed in the context of the calling task [8]. In the distributed situation, this would involve placing the code for the rendezvous on the processor holding the calling task. With the code

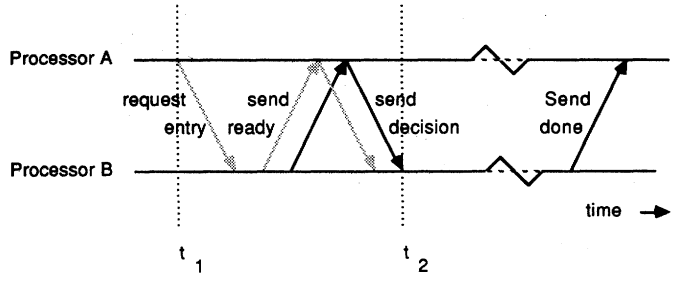


Fig. 7. Communication sequence for Case 4.

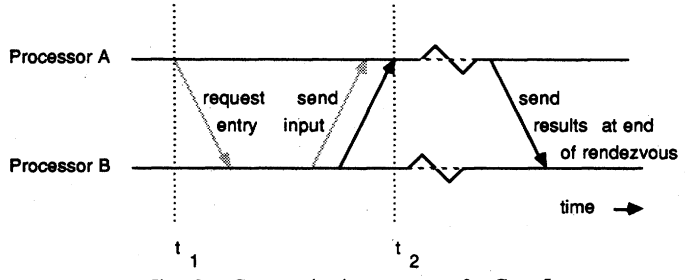


Fig. 8. Communication sequence for Case 5.

on the calling processor, it would seem that the only reasonable combination of the other parameters is to use I2 and make decision on the calling processor. This corresponds to Case 3 above. A message sequence for accomplishing this is shown in Fig. 8. Comparing this figure to Fig. 6, it can be seen that fewer messages are required, though at the expense of including any local variables of the called task as input and/or output parameters in the messages. Since in most cases the number of messages is more important in determining communication times than the length of the messages, this approach might have some advantages in terms of communication efficiency. However, this approach shares with Case 3 the ambiguity in interpreting the absence of receiving a response from the called processor by time t_2: one cannot tell if this is due to not reaching an appropriate **accept**, or a system failure.

E. Timed Entry Calls in the Presence of Timing Uncertainties

In most distributed situations the problem will be complicated, not only by a network delay, but by an uncertainty in the consistency of the sense of time maintained on two or more processors (see Section V for a detailed discussion of this point). Since two different machines will not have exactly the same value of time, it will not be possible to make a precise determination of whether the rendezvous can or cannot be started within the given time interval, as required by a strict interpretation of 1) and 2) above. From the perspective of the called processor, there will generally be a subinterval of measured time during which it is impossible to determine whether or not the specified delay has expired. See Fig. 9. A complete interpretation of time entry calls must state what is to be done if the called task becomes able to accept a call within this uncertainty interval.

An interpretation of timed entry calls that resolves this uncertainty is: "if the call can be guaranteed to be able to start within the given delay it is started and canceled otherwise." Thus, if the called task becomes able to accept a timed call

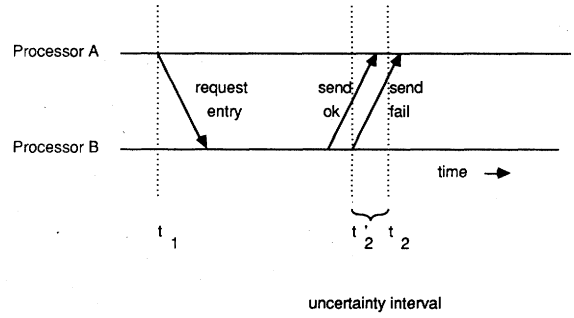


Fig. 9. Timed entry calls in the presence of uncertainty.

within the uncertainty interval, the entry call would be canceled even though in some instances it might actually have been within the specified delay; it is canceled because it is not possible to know that it is within the given delay.

F. Summary of Timed Entry Call Interpretations

There are thus several aspects to complete and consistent semantics of timed entry calls. For convenience, we summarize them here.

1) Timed entry calls imply a common sense of time between the machines holding the calling and called tasks.

2) The measurement of time for the expiration of the delay and the decision of whether or not to perform the rendezvous should be made on the machine holding the called task.

3) An implementation must guarantee that acceptance of a timed entry call means that the called task was ready to accept the call within the specified delay. The call fails if there is uncertainty about when the called task is ready to accept.

4) Exceptions should be used to handle timeouts caused by failures in system components.

In other words, we believe that the interpretation of timed entry calls given by Case 1 is the appropriate way to view interprocessor timed entry calls in a distributed system. Furthermore, exceptions should be used (rather than timed entry calls) to handle communication errors. Finally, a common sense of time is needed. The maintenance of a common sense of time between the calling and the called task is discussed next.

V. Maintaining a Network Sense of Time

It was noted in Section IV-A that the definition of timed entry calls implies a single sense of TIME throughout the execution of a program, and that it is not possible to absolutely achieve such a common sense of time across a distributed network. In this section, we consider various methods for managing distributed timing and discuss how to take their characteristics into account in the implementation of timed entry calls. We will show that the best that we can expect to do is to bound the differences in the sense of time on different processors in the system. The bound on the time synchronization among the processors will be treated as an additional uncertainty, as described in Section IV-E. Three methods will be considered, maintaining a network time server to which all processors go when they need a value for time, maintaining separate but synchronized clocks on each processor, and exporting the delay to be used on the called processor. This is

not intended to be an exhaustive list of methods; however, it is representative of the more obvious options available with current technologies.

A. A Network Time Server

The first mechanism we will consider is the use of a network time server. In this case each program use of a timing construct will require one or more accesses to the network server. The implementation of timed entry calls must take into account the time required to access the time server. We first describe the implementation scenario that could be followed for Case 1 and then develop an expression for the delay to be used by the called processor.

Referring to Fig. 2, the implementation sequence might be as follows:

- The processor containing the calling task will obtain the time from the network server and include both it and the specified delay in the timed entry call message sent to the processor holding the called task.
- The processor having the called task will call the network time server to obtain the time at the time the call is received.
- The processor containing the called task will compute the remaining time delay with which the called task is requested to start.
- Local management of the timed entry call will proceed as usual.

Thus, in addition to the network delay, there will be an effect from the time to make two accesses to the network time server.

Next we obtain an expression for the local time delay (d_l) to be used on the called processor to bound the time it will wait for an appropriate entry to be reached. For the purposes of this analysis denote the time measured on processor A by a superscript A, similarly for time measure on B. Further, let ΔT_{ns} to be the worst case difference in time that any processor can experience with respect to the server, i.e.,

$$t_j^i - t_j \leq \Delta T_{ns}$$

where i can indicate any of the processors in the system and t_j^i is the time returned by the time server when processor i makes the request at time t_j. In the case of an exact sense of network time, the local delay is given by

$$d_l = d - (t_a - t_1) \tag{1}$$

this guarantees the delay on B will not run past t_2. Taking server inaccuracy into account results in the following:

$$t_a < t_a^B$$

thus,

$$d - (t_a - t_1) > d - (t_a^B - t_1)$$

but

$$t_1 \geq t_1^A - \Delta T_{ns}$$

therefore,

$$d_l > d - (t_a^B - t_1^A) - \Delta T_{ns}. \tag{2}$$

Since t_a^B and t_1^A are the quantities that are measured (rather than t_a and t_1) the right-hand side of (2) is the best estimate we can obtain for d_l that guarantees that the called task is able to accept the call within the specified delay.

The utilization of a network time server is thus dependent upon our ability to bound the service time of the timer server. Two sources of service time must be considered, the propagation delay, and delays from interfering access requests. Propagation delays will depend upon the geometry of the system, and can often be bounded if there is suitable information about the geometry. However, the type of connection and not just its geometry must also be considered. Ethernets, for example, can not guarantee a bound; on the other hand, they might be acceptable in a practical sense. Delays due to interference of timer server requests from more than one processor may or may not be present, depending upon the particular method used to implement the network time server. If present, however, they usually inject an uncertainty in the response time from the server. If this cannot be bounded, then, strictly speaking, the network time server cannot be used as the basis for implementing timed entry calls in the distributed environment.

1) Use of Synchronized Clocks: An alternative method of providing timing is to maintain synchronism among the local clocks of the processors. There are then two issues to be considered here, the mechanism to be used to maintain correct Ada operation upon occurrence of a clock update and the development of an expression to be used for the local delay d_l on the called processor. We consider first the clock update.

a) Clock Update Correction: Until now, we have spoken of *a clock* on each processing unit, though the Ada semantics actually imply two, a time of day clock and a relative timer. We must be concerned with maintaining synchronism in both. For purposes of discussion, however, we will assume that we are talking about maintaining synchronism among a set of time of day clocks. Without further discussion we will assume that operations on these clocks are also reflected into the local relative timing clocks. Actually, as pointed out in [5], timing could be based solely upon the use of absolute timing, with an improvement in performance, though this is rarely done today.

The straightforward approach to the clock synchronization problem, and the one we will analyze, is to have a central master clock that periodically transmits time stamps to all of the local clocks so that they can be brought into agreement (synchronized). We will assume that the local clocks can drift with respect to one another and the master. This drifting can result in two situations when a synchronization time stamp is received: local time is either ahead, or behind of the time received. Let t^l be the time on the local clock when an update time stamp is received, and let t^s be the value of the time stamp received. When t^s is received, this value will replace t^l in the local clock. Depending upon the relative values of t^l and t^s different corrective actions must be taken.

In the first case, $t^l > t^s$. Resetting the local clock will essentially replay the local time for an amount of $t^l - t^s$. Thus, any local processes awaiting the expiration of a delay will have this amount of time added to their delay. For those that have been waiting since the last clock update, this will

simply compensate for the fact that the local clock was running too fast. For those which have been waiting for less than a clock synchronization period, the delay will be overcompensated slightly. In those cases corresponding the use of the **delay** only to delay a process, this does not matter, since Ada is only required to delay for at least as long as the specified delay. For situations where the delay relates to an interprocessor timed entry call, the delay has already been reduced by the synchronization uncertainty and again Ada semantics are maintained. However, in the case of local intraprocessor timed entry calls, the specific delay time could be exceeded. This indicates that the synchronization uncertainty must be taken into account even for local timed entry calls. The process for doing this is very similar to the establishment of the bound for d_l above, and will be discussed below.

In the second case, $t^s > t^l$. In this case, the update to the local clock bypasses the elapse of time on the local processor. By so doing, one or more scheduled delay expirations may be passed. It is thus necessary to check the list of scheduled delay actions and make ready any tasks whose delays expired during the clock update. It is worth noting that this happens automatically with the techniques described in [5] and no special checking of the schedule is necessary.

There are two obvious methods for distributing a master clock, by hardwired connection and by radio. Distributing the time signal by a wire is the easier to implement and perfectly adequate for many fixed location applications of moderate locality. (Microsecond precision is achievable if all systems are within 500 feet of each other and millisecond precision is achievable if they are within 100 miles). A physical connection between clocks of course is a potential source of failure, and limits applications to systems in which the components do not have autonomous mobility.

Synchronization can be achieved without connection by using one of the time keeping services supported by the U.S. Naval Observatory and the National Bureau of Standards [13], [14]. Both organizations provide phone line or satellite services; the NBS also provides a radio service (WWVB). These services are capable of providing time references with accuracy ranging from milliseconds to tens of microseconds, depending on the particular service and on the extent to which corrections are made for location. Clearly, receiving the signal and making corrections for location is more complicated and costly than dealing with a clock provided over a wire. On the other hand, in the case of satellites and radio stations no physical connections are needed.

An important question that arises is how frequently one must update the clocks in the system. This issue has been addressed empirically in [15] where it was found that to maintain synchrony within 10 ms on a collection of VAX computers, a synchronization process had to be executed once every 173 s. The clocks used in this case were not of high precision, however. Similar bounds would have to be established on the clocks to be used on each of the computers in the network.

b) Achieving Correct Ada Operation with Local Time Delay: Next we consider the use of multiple clocks, which are synchronzied periodically, for the implementation of Case 1. This is very similar to the network server case, except that instead of accessing a network server clock—thus adding delay times to time values received—we access a local clock whenever we need a time value. The local access can be much faster than access to the network time server, but the value returned has some error in it, as discussed above. The analysis to determine the lower bound that can be placed on d_l follows that for the network time server. Let ΔT_{sc} be the worst case difference in time between any clock in the system and the master clock, i.e.,

$$|t_j^i - t_j| \leq \Delta T_{sc}$$

where i can indicate any of the processors in the system, t_j^i is the time measured on the local clock and t_j is the corresponding time on the master clock. Then we have

$$t_a - t_a^B \leq \Delta T_{sc}$$

and

$$t_1^A - t_1 \leq \Delta T_{sc}.$$

These may be directly substituted into (1) to obtain

$$d_l > d - (t_a^B - t_1^A) - 2 \cdot \Delta T_{sc}. \tag{3}$$

As with the network server case, the right-hand side of (3) is the best estimate we can obtain for d_l that guarantees that the called task is able to accept the call within the specified delay.

While there are some similarities between the network time server case and the maintenance of synchronous clocks, there are important differences. With synchronized clocks the overhead of maintaining a network sense of time is decoupled from the use of the clock for timed entry calls. Thus, the overhead is not necessarily attached to timed entry calls, but is distributed over whole operation of the system. When the requested delay is large, the network time server case loses accuracy since the local relative clock used for timing the delay d_l may drift with respect to the server. In the analysis leading to inequality (2) an accurate local clock was assumed. With the use of synchronized clocks, this drift will not exceed the bounds derived since the local clock is periodically updated. Further, in the hardwired case it is often possible to keep the clock skew ΔT_{sc} much smaller than ΔT_{ns}. We thus prefer the synchronized clock method of maintaining a network sense of time.

2) Rely on the Exported Value of Delay: In some situations it may not be possible, or necessary, to share a common sense of time between processors (e.g., between satellites exploring deep space). In such cases, timed entry calls can be handled by exporting the time from the calling unit and use only this and local timing to manage things on the receiving processor. This requires knowledge of, or at least a bound on, the network communication times, and a bound on the relative drift of the local clocks. The implementation scenario for Case 1 is now a lot simpler than in the previous two cases: the processor containing the calling task transmits the specified delay to the called task. The best guarantee for d_l is now

$$d_l > d - \Delta T_{ex}$$

where ΔT_{ex} incorporates both the communication delay d_n and the relative drift. Unfortunately, in most cases that amount of drift grows with time and is unbounded. In practice, this is likely to place an upper bound on the length of delays that can be used, as in the network server case.

3) Uniprocessor Considerations: Considerations such as those described above can be carried out in a uniprocessor situation as well. For example, the delay d_n corresponds to the overhead associated with implementing the checking and rendezvous. Indeed, these times should be included in the ΔT's in the distributed situation as well. Depending upon the accuracy of the delay interval implemented, the ΔT's may be significant. This is likely to be the case for most processors at the 50 μs accuracy recommended in the RM and even more likely for the 10 μs accuracy discussed for some implementations. Strictly speaking, in these cases a timed entry call for small delays should fail even though a conditional entry call should succeed. This conformance is likely to be very difficult to measure, however.

VI. Summary and Conclusions

The need for distributed execution of Ada programs is growing rapidly as closely coordinated operation of multiple processors for such applications as robotics, space systems, and vehicle control increase. At this stage of development, distributed execution raises many issues of both interpretation and implementation. In this paper we have focussed on the impact of distributed execution on time-related constructs. Two constructs were singled out for attention because their effect can be interprocessor. These were the conditional entry call and the timed entry call. An anomaly with the timed entry call was pointed out that results from equating timed entry calls with zero or negative delay to conditional entry calls. Then, it was pointed out that there are several fundamentally different ways of interpreting timed entry calls across a network corresponding to the locations at which the time measurements and decisions are made. Placing both of these on the called processor causes fewer difficulties than the other choices. The use made of timed entry calls also affects the communication protocols necessary. It was recommended that the detection of network failures or device timeouts be associated with exceptions rather than the elapse of a delay in a timed entry call. It was also noted that the realization of a common sense of time across the distributed system is required, and an interpretation proposed that allows for a bounded variation in the value of time at different points across the system.

The possible interpretations presented are just that, possible interpretations. It remains for the governing bodies of the Ada Language to develope official interpretations of these constructs. It is hoped that this discussion will help in those deliberations.

References

[1] *Ada Programming Language (ANSI/MIL-STD-1815A).* Washington, DC 20301: Ada Joint Program Office, Dep. Def., OUSD (R&D), Jan. 1983.

[2] R. A. Volz, T. N. Mudge, A. W. Naylor, and J. H. Mayer, "Some problems in distributing real-time Ada programs across machines," *Ada in use, Proc. 1985 Int. Ada Conf.*, May 1985, pp. 72–84.

[3] R. A. Volz, T. N. Mudge, G. D. Buzzard, and P. Krishnan, "Translation and execution of distributed Ada programs: Is it still Ada?" to appear in *IEEE Trans. Software Eng.* Special Issue on Ada, 1987.

[4] M. C. Paulk, "Problems with distributed Ada programs," in *Proc. 5th Phoenix Conf. Comput. and Commun.*, 1986, pp. 396–400.

[5] R. A. Volz and T. N. Mudge, "Instruction-level mechanisms for accurate real-time task scheduling," in *Proc. IEEE 1986 Real-Time Symp.*, Dec. 1986, pp. 205–215.

[6] A. K. Mok, "The design of real-time programming systems based on process models," in *Proc. IEEE 1984 Real-Time Syst. Symp.*, Dec. 1984, pp. 5–17.

[7] L. Lamport, "Using time instead of timeout for fault-tolerant distributed systems," *ACM Trans. Programming Lang. Syst.*, vol. 6, pp. 254–280, Apr. 1984.

[8] A. N. Habermann and I. R. Nassi, "Efficient implementation of Ada tasks," Carnegie-Mellon Univ., Pittsburgh, PA, CMU-CS-80-103, pp. 1–21, Jan. 1980.

[9] A. Dapra, S. Gatti, S. Crespi-Reghizzi, F. Maderna, D. Belcredi, Natali, R. A. Stammers, and M. D. Tedd, *Using Ada and APSE to support distributed multimicroprocessor targets.* Commission European Communities, July 1982–Mar. 1983.

[10] J. Stankovic, "Achievable decentralized control for functions of a distributed processing operating system," in *Proc. COMPSAC 82,* Nov. 1982, pp. 226–230.

[11] K. Ramamritham and J. Stankovic, "Dynamic task sheduling in hard real-time distributed systems," *IEEE Software,* vol. 1, pp. 65–75, July 1984.

[12] R. Jha and D. Kafura, "Implementation of Ada synchronization in embedded, distributed systems," Dep. Comput. Sci., Virginia Polytechnic Inst. State Univ., Blacksburg, TR-85-23, pp. 1–56, 1985.

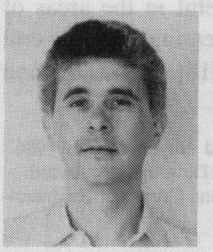
[13] G. M. R. Winkler, "Changes at USNO in global timekeeping," *Proc. IEEE,* vol. 74, pp. 151–154, Jan. 1986.

[14] R. E. Beehler and D. W. Allan, "Recent trends in NBS time and frequency distribution services," *Proc. IEEE,* vol. 74, pp. 155–157, Jan. 1986.

[15] R. Gusella and S. Zatti, "TEMPO a network time controller for a distributed Berkeley Unix system," *Distrib. Processing Tech. Committee Newslett.*, Informal publication of IEEE Computer Society Committee on Computer Processing, June 1984.

Richard A. Volz (M'60–SM'86) received the Ph.D. degree from Northwestern University, Evanstown, IL.

From 1964 to 1976 he was an Assistant Professor at the University of Michigan, Ann Arbor. He is now Director of the Robot Systems Division of the Center for Robotics and Intergrated Manufacturing at the University of Michigan. Prior to assuming this position, he held positions as Associate Director of the University Computer Center and Associate Chairman of the Department of Electrical Engineering and Computer Science. His current research includes the software/hardware computer architecture to support robot systems and the use of computer-aided design systems (CAD) for driving robot and sensor programming. Particular projects include CAD model-driven systems, automatic determination of grasp points (from CAD information), graphic programming of robots, and distributed systems integration languages for real-time control.



Trevor N. Mudge (S'74–M'77–SM'84) received the B.Sc. degree in cybernetics from the University of Reading, England, in 1969, and the M.S. and Ph.D. degrees in computer science from the University of Illinois, Urbana, in 1973 and 1977, respectively.

He is currently an Associated Professor in the Department of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor. He has been there since 1977. His research interests include computer systems architecture (both hardware and software), VLSI design, and computer vision.