LANGUAGE CONSTRUCTS FOR DISTRIBUTED REAL-TIME PROGRAMMING

Insup Lee
Vijay Gehlot

MS-CIS-85-58
GRASP LAB 55

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104

November 1985

Appeared in Proc. Real-Time Systems Symposium, Dec. 1985.

Acknowledgements: This research was supported in part by DARPA grants NOOO14-85-K-0018 and NOOO14-85-K-0807, NSF grants DCR-86-07156, DCR8501482, MCS8219196-CER, MCS-82-07294, 1 RO1-HL-29985-01, U.S. Army grants DAA6-29-84-K-0061, DAAB07-84-K-F077, U.S. Air Force grant 82-NM-299, AI Center grants NSF-MCS-83-05221, U.S. Army Research office grant ARO-DAA29-84-9-0027, Lord Corporation, RCA and Digital Equipment Corporation.

LANGUAGE CONSTRUCTS FOR DISTRIBUTED REAL-TIME PROGRAMMING

Insup Lee and Vijay Gehlot

Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

Abstract

For many distributed applications, it is not sufficient for programs to be logically correct. In addition, they must satisfy various timing constraints. This paper discusses primitives that support the construction of distributed real-time programs. Our discussion is focused in two areas: timing specification and communication. To allow the specifications of timing constraints, we introduce the language constructs for defining temporal scope and specifying message deadline. We also identify communication primitives needed for real-time programming. The issues underlying the selection of the primitives are explained, including handling of timing exceptions. The primitives will eventually be provided as part of a distributed programming system that will be used to construct distributed multi-sensory systems.

Introduction

For many computer applications, such as robot arm control, missile control, on-line process control, etc., it is not sufficient for programs to be logically correct. In addition to being logically correct, the programs must satisfy certain timing constraints determined by the underlying physical process being controlled to avoid possible catastrophic results. Programs whose correctness depends on the adherence of timing constraints are called real-time programs. Furthermore, it has been recognized that for certain real-time applications, such as multi-sensory robot systems [12], it is natural to distribute control among several connected processors. This distributed view of the underlying system allows real-time programs to be implemented as relatively independent processes which run asynchronously except for occasional synchronization and communication. We call such concurrent programs which are required to respond to external and internal stimuli within a specified deadline distributed real-time programs.

The conventional approach to real-time programming has been to write a concurrent program to be logically correct while ignoring real-time constraints [3, 25]. The scheduling primitives are then added by the programmer to satisfy real-time constraints after the program has been shown to be logically correct. For example, Modula-2 provides *transfer* procedure that can be called from a program to switch process [26]. The language proposed by Berry et al. allows the programmer to write a scheduler separately from a program to control process switching [3]. The disadvantage of these two approaches is that the programmer has to figure out and implement a scheduling strategy that satisfies the timing constraints of a program. Although there have been many languages designed for distributed programming, they do not allow the specification of timing constraints except delay or sleep, and timeout. If such languages are used for real-time programming, it is impossible to understand whether or not a given program has to satisfy any timing constraints from the program text. As Allchin points out [1], it is also difficult and awkward to write programs with timing constraints using a language that does not explicitly support the notion of time. We believe that a language designed for distributed real-time programming should allow the specifications of timing constraints. Furthermore, the underlying scheduler should use timing information to schedule processes so that as many timing constraints as possible are satisfied. That is, the responsibility of scheduling should be left to the system, removing the burden from the programmer.

In the next section, we explain the motivation behind the design of language constructs for distributed real-time programming. Section 3 explains the assumptions and basic model of our system. Section 4 identifies timing constraints useful for real-time programming and explains how they are specified in our system. Section 5 describes interprocess communication primitives designed for synchronization and communication among real-time processes. Our emphasis is on how to control buffer overflow and how to specify timing constraints. The last two sections contain an example and discussion on future work, respectively.

Motivation

The language constructs we describe in this paper are to be included in DPS to facilitate distributed real-time programming. The motivation behind the development

of DPS (Distributed Programming System) is to provide an easy to use programming environment for the construction of a distributed program from sequential programs written in C, LISP, and Prolog [11]. The backbone of DPS is a distributed configuration specification language (DICON) which is used write the configuration specifications of a distributed program describing resource requirements, process interconnection, and process assignments. DICON supports nested configuration specifications to allow the modular construction of a distributed program. The prototype system of DPS without real-time capability has been built and is being used for the development of multi-sensory systems.

The major design goal is to provide language constructs that can be used to specify the timing constraints of code execution and interprocess communication. Rather than designing a complete language, we extend an existing language, namely C, with primitives suitable for distributed real-time programming. Although our system is to support the development of multi-sensory systems, it is general enough to be used for other real-time applications.

Another design goal is to allow the detection and handling of exceptions caused by timing constraint violations at run-time. Although it is desirable to assume the adherence of all timing constraints during execution, we believe that such an assumption is not valid in practice. For example, real-time systems might miss deadlines due to some hardware failure or unexpected environmental interference even if software is assumed to be correct, which is rarely (if not never) true. So, languages designed for real-time programming should provide mechanisms to detect and handle timing exceptions.

The last design goal is that the run-time support system of the language should use timing information specified within a program to schedule processes. It is conceivable to treat specified timing constraints as assertions that a program has to satisfy during its execution and to require the programmer to come up with a scheduling strategy (e.g., static priority) independent of timing specifications that is likely to meet all the timing constraints. We believe that processes can be scheduled more efficiently when their timing constraints are considered and that scheduling should be left to the underlying system.

Although many languages have been designed for real-time programming, most of them do not allow the specifications of timing constraints within a program [1, 7]. The notable exceptions are PEARL [14] and ESTEREL [4]. PEARL has been successfully used to implement a wide range of real-time applications for a single processor system. It, however, seems that PEARL cannot easily be extended for distributed systems and does not allow exception handling for missed timing constraints. In ESTEREL, time is considered as flow of events; so, all temporal constructs are event-based. It does provide an exception handling mechanism. It is not clear how processes are scheduled and the designers of ESTEREL make an unrealistic assumption that message transmission is instantaneous.

Assumptions and the Basic Model

The distributed program of interest runs on processors that are connected by a communication network. The processors communicate with each other only through the network; in particular, there is no shared memory. We assume that all clocks on the network are synchronized within a fixed small time interval [8, 10, 15]. Granularity of timing constraints involving different nodes is assumed to be large compared to the discrepancy among the clock values of different nodes.

A distributed real-time program is viewed as consisting of a set of internal processes, external processes, and shared objects. An (internal) process is defined by the programmer and represents a logically independent execution thread of control. An external process represents part of the external world which the distributed real-time program is to interact with and control. An object is an instance of an abstract data type and provides operations that can be invoked by processes. Processes communicate and synchronize with each other by exchanging messages. A device object represents the abstraction of attached special purpose hardware and consists of interrupt and control routines. These routines are invoked by internal and external processes. Interrupt routines invoked by an external process may send messages to other processes for further processing. Each internal process includes all timing constraints that have to be satisfied during its operation. These timing constraints are with its code segment or with messages it sends.

In our system, a distributed program is configured off-line and that all processes are created when the program starts executing. That is, we do not allow the dynamic creation of real-time processes. We believe that this is not a severe restriction for real-time programming, at least for multi-sensory systems as we see now. The execution of a distributed real-time program consists of two phases: initialization and operation. The first phase is carried out by the main process of a distributed program which invokes the initialization routines of component processes in the order defined by the programmer and then schedules them for the operation phase. When the first phase is completed, the operation phase starts. At that time, the main process becomes an idle process waiting for global timing exceptions.

We note that our approach is not necessarily to support the fast execution of a distributed program. Rather, it is to allow the programmer to develop a program knowing exactly what happens when. Our eventual hope is to be able to verify the correctness of real-time programs.

Timing Specifications

A process that has timing constraints and whose correctness depends on whether its timing constraints are satisfied is called a real-time process. There are two kinds of real-time processes: periodic and sporadic [16]. A periodic process becomes ready at regular intervals and a sporadic process becomes ready at any time. The timing constraints of a process are defined on the whole process or a part of the process. Requirements for real-time processing can be viewed as when certain processing has to take place for how long and how soon. There are two ways to represent time in our system: relative time and absolute time. The absolute time refers to a wall-clock time (e.g., Eastern Standard Time) and is represented by (year:month:day:hour:min:sec:msec). The relative time is specified in units of hours, minutes, seconds, and milli-seconds and is used to specify timing constraints relative to the current time, called *now*.

The kinds of timing constraints useful for the requirement specifications of real-time programs are as follows [6]:

1. Maximum - No more than t time units may elapse between two events.
2. Minimum - No less than t time units may elapse between two events.
3. Duration - An event or a sequence of events must occur for t time units.

An event is any action that can change a program state; e.g., execution of a statement, sending of a message, receiving of a message. Timing constraints can be originated from a process itself, a communicating process, or the external world.

To facilitate the programming of the above three kinds of timing constraints, we provide language constructs called temporal scopes which identify a sequence of statements, possibly empty, with timing constraints. The possible attributes of a temporal scope are as follows:

- deadline - the latest time in which the execution of a temporal scope can be completed.
- minimum delay - the minimum amount of time that should pass before starting the execution of a temporal scope. The minimum delay is used to specify how long a process should sleep.
- maximum delay - the maximum amount of time that should pass before starting the execution of a temporal scope. This delay is used to specify how long a process is willing to wait for a message to arrive.
- maximum execution time - the maximum computation time necessary for the execution of a temporal scope. This is to ensure that the statements within a temporal scope do not execute longer than the required amount of time.
- maximum elapse time - the maximum execution time plus all user-defined delay during the execution of a temporal scope. That is, the maximum elapse time specifies how long it would take to execute the statements of a temporal scope assuming that they are executed as soon as possible. The specification of the maximum elapse time gives the scheduler information needed to find scheduling that satisfies timing constraints when the earliest-deadline-first scheduling fails.

There are three kinds of temporal scopes: global, local, and communication. A global temporal scope encapsulates a whole process and is used to define a periodic process. Local temporal scopes are used to specify timing constraints within a process. Communication temporal scopes are used to specify the timing constraints associated with interprocess communication.

In the most general form, a local temporal scope allows the programmer to specify when a sequence of statements should start executing for how long and how soon. The (approximate) form of the internal temporal scope is as follows:

```
start <d-part> [ <e-part> ][ <dl-part> ] do
    <start-body>
[ <exceptions> ]
end
```

The meaning of the construct is that the current process is delayed as specified by <d-part> and then the statements of <start-body> are executed. The execution must be completed within <dl-part>, the deadline. Furthermore, the total execution time of the statements should be less than the amount of time specified in <e-part>. If any of these timing constraints is not met, the execution of the construct is terminated and an exception is raised. The raised exception is handled within a handler provided at the end of the construct if it is provided; otherwise, the exception is ignored. In the latter case, the execution resumes with a statement following the construct. We discuss exception handling in Section 6.

The delay-part of the construct can be specified using an absolute time or relative time as follows:

```
<d-part>::= now| at <abs-time>| after <rel-time>
```

"now" means that there is no delay associated with the construct. "at <abs-time>" means that the execution of <start-body> should start at a given wall-clock time. "after <rel-time>" means that the execution of <start-body> should start after waiting for the time units specified in <rel-time> from the current time.

The optional execute-part defines the maximum amount of execution or elapse time that is required to

complete the construct. It is specified as follows:

```
<e-part>::= execute <rel-time>| elapse <rel-time>
```

The execute time includes only the amount of time needed to execute <start-body>, whereas the elapse time includes the amount of time to be spent waiting within <start-body> in addition to its execution time. If a specified timing constraint is violated, the execution of the construct terminates and an execute-time or elapse-time exception is raised. If the execute-part is omitted, an execute-time or elapse-time exception is never raised.

The deadline-part defines how soon the execution of the construct has to be completed. As with the delay-part, a deadline can be specified using absolute time or relative time and its syntax is as follows:

```
<dl-part>::= by <abs-time> | within <rel-time>
```

If relative time is used, it is relative to when the construct starts executing; that is, immediately after the delay-part is completed.

As an example, a process can be put into sleep for 10 seconds as follows:

```
start after 10 sec do end
```

An example with a delay-part and a deadline-part is as follows:

```
start at (9h:00m) within 10 sec do
   /* statements to be executed when woken up */
exception
   /* exception handler for missed deadline */
end
```

It means that the process should be delayed until nine o'clock and the statements within the construct should be executed between nine o'clock and 10 seconds after nine o'clock.

Another kind of the temporal scope is the communication temporal scope. It is used to specify timing constraints associated with interprocess communication. The following timing constraints may be specified:

1. how soon a message should be received and processed by a receiving process after it is sent.
2. how long a sending process is willing to wait for a reply, if any, after a message has been sent.
3. how long a receiving process is willing to wait for a message to arrive.
4. how long it takes a receiving process to process a message after it has been received.

In our system, (3) and (4) are specified within a receiving process, whereas (1) and (2) are defined by a sending process since a receiving process cannot determine the deadline without first receiving a message. If any specified timing constraint is violated, an exception is raised and handled by an exception handler attached to the enclosing local temporal scope. The communication temporal scope is explained in detail later.

The following defines the general form of yet another type of temporal scope called the repetitive temporal scope.

```
from <start_time> to <end_time> every <period>
execute <exec_time> within <deadline> do <stmts>
[<exceptions>]
end
```

Informally, the meaning of the construct is as follows: Starting at <start_time> <stmts> are executed periodically with a period =<period> and a deadline = <deadline> until <end_time>. <start_time> and <end_time> may either be constant or variable.

We distinguish one more type of temporal scope called consecutive temporal scope. Its general form is as follows:

```
cstart <delay_1> [<execute_1>] [<deadline_1>] do
        <stmts_1>
[<exceptions_1>
cstart <delay_2> [<execute_2>] [<deadline_2>] do
        <stmts_2>
[<exceptions_2>]
        ...
cstart <delay_n> [<execute_n>] [<deadline_n>] do
        <stmts_n>
[<exceptions_n>]
end
```

In general, <$delay_i$>'s are not related nor are <$execute_i$>'s and <$deadline_i$>'s. Thus, consecutive temporal scope is a composite temporal scope consisting of finite sequence of unrelated temporal scopes and hence the meaning of each of the subscopes is independent of the others.

We allow temporal scopes be nested but not overlapped. Inconsistent deadline specifications are ignored at run-time. For example, a nested temporal scope with a deadline which is later than that of its enclosing temporal scope does not change the deadline of the process. Inconsistent deadlines should be detected at compile-time as much as possible; however, they cannot be detected if deadlines are expressed using run-time expressions.

In general, the timing constraints of a sporadic process can be specified using local and communication temporal scopes. They are, however, not sufficient to specify periodic processes since the timing constraints of a periodic process should not depend on when processes are executed. Otherwise, if a periodic process misses its period, it may never be scheduled properly again unless there is a way to determine how many periods it has

missed so far. Furthermore, whether or not a process is periodic should be stated explicitly to help the scheduler and the timing verifier since a periodic process has less demand on resources than sporadic process with similar timing constraints [17]. A periodic process is scheduled using the schedule command whose arguments are the process name, start and end time, period, optional execution time, and deadline. For example, a periodic process Stir is scheduled to be executed for two seconds within five seconds at every 10 second interval starting from the current time plus one minute for the next twenty minutes as follows:

```
schedule Stir at now+1min every 10sec
  execute 2sec within 5sec until now+20min
```

After the main process schedules all periodic processes, it becomes an idle process which handles exceptions caused by periodic processes. For example, if a periodic process missed its period or deadline, an exception is raised and handled by a handler provided within the main process. Here, a periodic process can be unscheduled and then scheduled again with different timing attributes.

Communication

Processes communicate by sending and receiving messages. Reasons for sending messages in real-time systems can be distinguished as follows: to forward data or signal to another process, to synchronize with another process, or to request an action from another process. The first case involves sending messages only, whereas the second and third cases require sending messages and then receiving replies. In the first case, a sending process need not be blocked. Thus the most basic communication primitive we provide is *send-no-wait*. In the second case, a sending process is blocked until a receiving process is synchronized as in CSP [9]. In the third case, a sending process should be blocked only when a reply is needed to continue its execution rather than immediately after the send as with Ada[1] *rendezvous* and remote procedure call [5, 13, 23]. For example, a process may send a message before its reply is needed to reduce or eliminate the amount of time it has to wait for a reply. Although the second and third cases can be emulated by a pair of *send-no-waits*, the resulting program becomes complex and hard to understand. Furthermore, it is not easy to emulate many-to-one communication if a receiving process has to explicitly figure out which process has sent a message in order to reply. In addition to one-way asynchronous communication, our system supports two-way communication in which the destination of a reply need not be explicitly mentioned.

A message can be received either explicitly or implicitly [21]. In explicit receive, a process deliberately receives a message by executing a receive operation. Here, a receiving process should be able to specify how long it is willing to wait for a message to arrive and what to do with a tardy message. In implicit receive, a procedure-like body of code is activated automatically by the arrival of an appropriate message. Thus, there is no timeout associated with implicit receive. We believe both the kinds of receive should be supported as some applications are programmed naturally with the former and other applications with the latter. Which receive is more appropriate usually depends on a hierarchical relation between a sending and receiving processes [22].

Although numerous communication primitives have been developed for distributed programming [2, 21, 22], none of the existing designs allow the specifications of timing constraints, except timeout [24, 13]. Our communication primitives are designed to support the specifications of timing constraints and the detection and handling of exceptions raised by missed timing constraints. Besides timing specifications, another important issue is that of buffer overflow control. For real-time communication, there should be no unexpected delay caused by the overflow of message buffers. We explain how buffer overflow is controlled in our system. Another important issue in developing communication primitives is a message type checking issue; it has been discussed elsewhere [11] and will not be repeated here.

Naming and Buffer Control

In order to communicate, processes need to be able to name each other. Names may be established at compile-time or created at run-time. The main advantage of dynamic creation of names is flexibility; but, it complicates the static analysis (e.g., deadlock detection) of a distributed program. In our system, names are established at compile-time as our goal is to be able to verify timing constraints statically. The timing verification becomes impossible if it is not possible to determine which processes communicate with each other.

Naming can in general be either direct or indirect [22]. The direct naming can be 1-way or 2-way. The indirect naming is based on port or link. In our system, messages are sent to and received from links. There are two kinds of links: one-way link for unidirectional communication and two-way link for bidirectional communication. Each end of a link is named locally within a process using a port. Each port has a unique identifier that distinguishes it within a process; however, the same port identifier can be used in different processes. Permitted link types are one-to-one, one-to-many, and many-to-one. Processes are implemented using local ports and links between local ports of communicating processes are defined within a configuration specification written in DICON [11]. The two advantages of our naming approach are as follows: (1) it is easier to combine separately developed sequential programs without having to worry about port naming conflicts; and (2) buffer overflow control strategies, which depends on the characteristics of sending and receiving processes, can be defined with link

[1] Ada is a registered trademark of the U.S. Government

declarations when sequential programs are combined into a distributed program.

Unlike the communication paradigm for distributed programs, where every message sent is expected to be received by a receiving process, there are three kinds of real-time communication paradigms:

1. Asynchronous communication with non-queued message - In this paradigm, processes execute asynchronously. A process sends a message to another process and the message is never queued. So, if the other process is not waiting for the message, it is lost.
2. Synchronous communication without message loss - In this paradigm, processes execute synchronously and execution is coordinated by the acceptance of a message and a (possibly null) reply.
3. Synchronous and asynchronous communication with possible loss of aged message [18, 20] - In this paradigm, process interaction assumes that a fixed number of recent messages of one process is available to other processes. Message loss results from message buffer overflow or the expiration of a message deadline.

Message loss should not be caused by the unreliable nature of the underlying communication medium; rather, it should be the characteristics of communicating real-time processes. If the loss of message is anticipated, a distributed real-time program must be structured to function correctly with occasional loss of messages.

To support the three communication models, we allow the programmer to specify the deadline of a message and the size of message buffers for each link and its overflow control strategy as whether to keep the last N or the first N messages. The size of message buffers is statically fixed and cannot be changed dynamically to avoid unexpected delay due to the lack of buffer space. We do not support the blocking of a sending process when there is no available buffers as an option. Adding this feature would require that a sending process be blocked after every send until an ack is received from a processor on which a receiving process resides. This blocking happens for every send since it may not be possible to determine a priori whether or not there is an available buffer at a receiving end without actually sending a message.

Unidirectional Communication

The most basic communication paradigm supported in our system is an asynchronous communication using a one-way link. A one-way link is established by connecting an out-port to an in-port. An out-port is used to send a message and an in-port is used to receive a message. A sending process sends a message by calling a send operation with a variable containing a message as follows:

```
Send (OutPortId, var)
```

After the call, the sending process resumes immediately.

A process receives a message by executing (approximately) the following accept statement:

```
Accept on <port-list> [within | by] <timeout>
   when Port1 (arg) : /* statements */
   when Port2 (arg) : /* statements */
   ...
   when Portn (arg) : /* statements */
   when TimeOut : /* statements to handle timeout */
end Accept
```

The meaning of the statement is as follows: If messages have already arrived at ports in the port list, the most time critical message is removed first. (A way of specifying timing constraints with a message will be explained shortly.) Then, the associated statements are executed. If no messages are waiting, the process waits for a message or times out, whichever happens first. In the latter case, the process executes a timeout exception handler if provided.

An accept construct defines a communication temporal scope. Possible timing constraints with unidirectional communication are message deadline (i.e., how soon a message must be received and processed) and message processing time (i.e., the maximum time to process the message). The former is specified with an out-port declaration as it can best be determined by the sender of a message. The latter is specified with an in-port declaration as it is the property of a receiving process. The consistency of timing constraints of the linked out-port and in-port is checked when they are linked at compile-time. The slack between message deadline and processing time defines the deadline of message delivery. An exception raised by missed deadline or too much processing time is caught by an handler provided at the end of the enclosing local temporal scope. Exception handling is discussed in Section 6.

One limitation of *send no-wait* is that a sender will not be notified when a message is not received on time. This unexpected delay might be due to the contention or failure of hardware or the unwillingness of a receiving process. We provide two options to the programmer to handle a message whose deadline expires before it is read by a receiving process. The first is to define an in-port to receive only messages whose deadlines have not yet passed. Here, messages with expired deadlines are dropped by the run-time support system. The second is to keep aged messages as long as there are enough buffers and to raise a deadline exception when a message with an expired deadline is received. If a sending process needs to know the success or failure of message reception, it can do so using the bidirectional communication primitives described in the next section.

In real-time programs, it sometimes is necessary to send a message at known future time rather than now. Such a message can be sent using the following

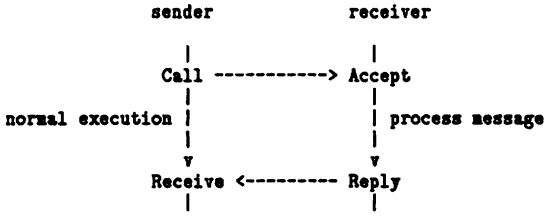
operation:

```
DelayedSend (OutPortId, time, var)
```

Its effect is as if a normal send call has been executed at a specified time. The number of outstanding time delayed messages is subject to the buffer size of a link on which an out-port connected. Since a receiving process has to explicitly wait for a message to arrive, we do not provide a delayed receive operation as it can be achieved by putting the process to sleep until the desired time.

Bidirectional Communication

Another form of communication is a pair of asynchronous communications on a two-way link. A two-way link is established by connecting a call-port and an entry-port. A call-port is used to send a message and to receive a reply, whereas an entry-port is used to receive a message and to send a reply. A typical scenario of bidirectional communication is as follows:



A sending process executes a Call operation to place a message into a port and then continues until a reply for the message is needed. The message is delivered to the receiver and causes the receiver's execution of an Accept operation to be completed. The receiver handles the message and then sends back a reply. The reply causes the sender's execution of a Receive operation to be completed.

Possible timing constraints with bidirectional communication are (1) how soon a reply should arrive to a sending process after a message has been sent (deadline of a message) (2) how long a receiving process is going to execute to produce a reply after a message has been received (processing time of a message) (3) how long a receiving process is willing to wait for a message to arrive. The deadline of a message is defined with a call-port declaration and a maximum processing time is defined with an entry-port declaration. Timeout value is specified with the Accept construct as it depends on when the construct is executed. From (1) and (2), the configurator can determine the deadlines of a message and a reply delivery. Each delivery deadline is the half of the difference between the message deadline and processing time.

The approximate syntax of the call statement is as follows:

```
Call (CallPortId, msg)
   /* statements */
Receive (CallPortId, ArrayVar, NumOfReplies)
```

The meaning of the statement is as follows: A process sends a message and then resumes its execution until replies are needed. If less than the number specified in NumOfReplies of replies have arrived, the process waits for more replies. If the current deadline is missed while waiting for more replies, the process executes the deadline exception handler defined with the enclosing temporal scope as discussed in Section 6. A Receive call includes an array variable for replies and specifies how many replies the process is expecting. This generality is needed to support a two-way link with many receivers. The syntax and meaning of the Accept statement is the same as those for the one-way link except that the last of the statements associated with an entry-port must be a Reply statement.

We note that Ada's *rendezvous* construct can be emulated by placing a Receive operation immediately after a Call operation and that synchronous communication (similar to *send-wait*) can be achieved by placing Receive and Reply immediately after Call and Accept, respectively. Send-wait normally means that a sending process is blocked until a message is received. The difference between send-wait and synchronous communication is that the run-time system can take care of sending an ack to unblock a sending process in the former, whereas a receiving process has to explicitly send a reply in the latter. Thus, send-wait can be implemented with less run-time overhead. We, however, believe this saving is rather minuscule. Also, send-wait and multiple destinations do not mix very well. For example, how long should a sender be blocked. A sender might be blocked until the message is received by all the receivers or by any process. If timeout is allowed, it may not be easy (or is awkward) to figure out which processes have received the message.

Communication with Shared Objects

In our system, a shared object (e.g., data, devices) is supported as a set of procedures that can be called from other processes. A procedure of an object is invoked by sending a message to a port which is linked to the procedure. The syntax of remote procedure call is the same as that of sending a message and receiving a reply. Each object declaration identifies a set of procedures that can be invoked from other processes as in-port and entry-port depending on whether or not they return values. So, a link between a port and a remote procedure can be either one-way or two-way. An invoked procedure returns a value through the execution of an explicit return statement. As before, the deadlines of remote procedure calls are specified with out-port and call-port declarations, whereas the maximum execution times of remote procedures are specified with in-port and entry-port declarations.

Unlike explicit message receive, the order in which remote procedure calls are executed is assumed to be not important. So, to satisfy the deadlines of as many remote procedure calls as possible, pending remote calls are executed in the order of earliest modified deadlines.

A modified deadline is deadline minus execution time if the latter is specified; otherwise it is equal to deadline. In our system, at most one process is dedicated to handle all remote procedure calls on an object, as creating or allocating a different process for each call will not improve the response time of the calls. To be able to meet the deadline of a more urgent request, it is possible to preempt the current procedure execution and to handle another remote call. Since the concurrent activations of operations can result in an inconsistent state for the object, we allow the programmer to specify the procedure activations which a procedure call can preempt. Thus, each procedure contains a list of procedures that can preempt its activation. This information is also used to start the next call with a new stack while the execution of the current call is waiting for a reply from another node even if the deadline of the next call is later than that of the current call. This is permitted only if both can preempt each other. It is the programmer's responsibility to preserve the consistency of an object state if concurrent procedure activations are allowed.

Exception Handling

The language incorporates an exception handling mechanism in order to cope with timing errors. Issues in providing exception handlers for timing errors are as follows: (1) when an exception should be detected; (2) which processes should handle the exception; (3) what recovery actions are meaningful and possible; and (4) how soon should an exception be handled. Exception handlers can be attached at the end of the local temporal scope and the body of the main process. If a local temporal scope contains call and accept statements, their exception handlers are also appended at the end of the enclosing local temporal scope. Thus, the syntax is

```
start ...
   ...
exception
   when <exception list> within <deadline> : ...
   when <exception list> within <deadline> : ...
   ...
end
```

A handler (when-clause) is executed in place of the unexecuted portion of code within the current temporal scope when an exception is detected.

Built-in timing exceptions are provided for the following conditions:

- Failure to complete the temporal scope in time (deadline exception).
- Failure to complete a call or accept construct in time (port-id.deadline exception).
- Attempt to execute the temporal scope longer than a specified maximum execution time (execution time exception).
- Attempt to execute an accept construct longer than a specified maximum execution time (port-id.execution time exception).

When the deadline or maximum execution time of a temporal scope is violated, the execution of a process is stopped immediately if it is executing. However, its handler should be executed within the deadline specified with the handler. Possible actions within an exception handler is either to resume with modified deadline and/or maximum execution time or exit the scope and resume. If another exception is raised while handling an exception, the current exception is nullified and the new exception is handled.

An exception raised due to the missed deadline or execution time violation of a periodic process is handled by an exception handler associated with the main process. The possible recovery actions are to resume, to skip this period, or to reschedule with different timing constraints.

When a message is sent using a one-way link, no timing exceptions are possible with a sending process. There are, however, three possible timing exceptions with a message and a receiving process. If a message does not arrive at a receiving process in time, the receiving process times out and starts its exception handler if provided. After receiving a message, if a receiving process cannot process the message within a deadline, it causes a deadline exception. If a message is not delivered or received in time, the programmer has an option as to throw away the message or to leave it in the queue of a receiving process. In the latter case, a deadline exception is raised when a receiving process read the message.

When a message is sent using a two-way link, a sending process starts a temporal scope with a deadline. Thus, if the scope is not completed within the deadline, the sending process is forced to execute the handler associated with the deadline exception. Possible exceptions for messages and receiving processes are same as those described in the previous paragraph. Replies that are not received within their deadlines are dropped.

An Example

As an illustration, we consider an automated kitchen equipped with two physical devices, viz. an oven and a range with on/off control. The kitchen is operated by a cooking robot and the problem is to bake a chicken for 15 minutes and to make a stir-fry which takes 10 minutes. Both these dishes must be finished within 20 minutes. Also, while making the stir-fry, the wok has to be stirred for at most 10 seconds every 40 second interval.

We assume that the oven provides two in-ports, OvenOn and OvenOff, and that the range provides two entry-ports, RangeOn and RangeOff. The cooking robot is programmed as follows:

```
process cooking_robot;
   call-port RangeOn [deadline 2 sec];
   in-port OvenOn [deadline 2 sec],
           OvenOff [deadline 2 sec];
   var ToBeDone : time
begin
   start now within 20 min do
     call (RangeOn,nil); /* turn on the range */
     send (OvenOn, nil); /* turn on the oven */
     receive (RangeOn,nil,1);/* wait for range on */
     /* put chicken in the oven and cook */
     ToBeDone := now + 15 min;
     delayedsend (OvenOff, ToBeDone, nil);
     /* Move stuff into the wok and cook */
     from now to now+10 min every 40 sec
     execute 10 sec within 10 sec do
        " Stir " /* stir for 10 sec every 40 sec */
     end;
     /* Turn off the range */
     call (RangeOff, nil);
     receive (RangeOff,nil,1);
     /* wait until the oven is turned off */
     start after (ToBeDone-now) do end;
   end;
end;
```

Discussion

This paper discussed issues that arise in the design of language constructs supporting distributed real-time programs. We have identified two areas: timing specification and communication. To support timing specifications, a novel construct called a temporal scope has been proposed. A temporal scope allows the programmer to specify timing constraints and exception handlers to cope with timing errors. Allowed timing constraints are meant to be general and complete enough for a wide range of hard real-time applications. We permit timing constraints to be associated with code execution and communication. Temporal scopes are designed to support nesting but not overlapping to facilitate timing analysis at compile-time.

In the area of communication, we have discussed what primitives are needed for real-time programming. We chose asynchronous and bidirectional communication primitives that impose as little blocking as possible to sending processes. We also discussed various aspects of communication, including buffer overflow control and message deadline.

The paper discussed our current design on issues under study; the design is likely to be modified as our understanding increases. There are issues that we have not addressed in this paper. For example, given timing constraints, what is the best way to schedule real-time processes. It seems that the earliest modified deadline first algorithm is a realistic way of scheduling real-time processes, where the modified deadline of a temporal scope is equal to the deadline minus the maximum execution time, if the latter is specified; otherwise, it is equal to the deadline. In this way, the programmer can control the scheduling of processes for temporal scopes with internal delays. This algorithm is a variation of the "preemptive least processor time to go" which has been shown to be better than the earliest deadline first algorithm when scheduling blocks containing internal delays [19].

Other issues that we have not addressed are as follows: Can we automate the allocation of processes into processors based on timing constraint information specified with processes? Is it possible to verify that all timing-constraints will be satisfied statically? We are currently investigating these issues and are sure that insights gained by our study will provide us with positive feedback to the refinement of the design. The last issue remained to be addressed is whether or not the proposed constructs make distributed real-time programming easier. Our plan is to gain experience by writing a number of distributed real-time programs and to build a run-time system on a network of MicroVAX II's.

Acknowledgements

The authors gratefully acknowledge the contributions made by members of the GRASP group; in particular, Robert King and Gaylord Holder. This research was supported in part by NSF DCR 8501482, NSF MCS-8219196-CER, ARO DAA6-29-84-k-0061, AfOSR 82-NM-299, NSF MCS 82-07294, and AVRO DAABO7-84-K-FO77.

References

1. Allchin, J.E. "Modula and a Question of Time". *IEEE Tran. on Soft. Eng. SE-6*, 4 (July 1980), 390-391.

2. Andrews, G.R., F.B. Schneider. "Concepts and Notations for Concurrent Programming". *Computing Surveys 15*, 1 (March 1983).

3. Berry, D.M., Ghezzi, C., Mandrioli, D., and Tisato, F. "Language Constructs for Real-Time Distributed Systems". *Computer Languages 7*, 1 (1982), 11-22.

4. Berry, G., Moisan, S., and Rigault, J.-P. ESTEREL: Towards a Synchronous and Semantically Sound High Level Language for Real-Time Applications. Proc. Real-Time Systems Symposium, IEEE, 1983, pp. 3-19.

5. Birrell, A.D. and Nelson, B.J. "Implementing Remote Procedure Calls". *ACM Trans. on Computer and Systems 2*, 1 (Feb. 1984), 39-59.

6. Dasarathy, B. "Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them". *IEEE Tran. on Soft. Eng. SE-11*, 1 (Jan. 1985), 80-86.

7. Gligor, V.D. and Luckenbaugh, G.L. An Assessment of the Real-time Requirements for Programming Environments and Languages. Proc. Real-Time Systems Symposium, IEEE, 1983, pp. 3-19.

8. Gusella, R. and Zatti, S. TEMPO - A Network Time Controller for a Distributed Berkeley UNIX System. Proc. USENIX Conference, 1984, pp. 78-85.

9. Hoare, C.A.R. "Communicating sequential processes". *CACM 21*, 8 (August 1978), 666-677.

10. Lamport, L. "Time, Clocks, and the Ordering of Events in a Distributed System". *Comm. of the ACM 21*, 7 (July 1978), 558-565.

11. Lee, I. A Programming System for Distributed Real-Time Applications. Proc. Real-Time Systems Symposium, December, 1984.

12. Lee, I. and Goldwasser, S.M. A Distributed Testbed for Active Sensory Processing. IEEE Int. Conf. on Robotics and Automation in St. Louis, Missouri, March, 1985.

13. Liskov, B., R. Scheifler. "Guardians and Actions: Linguistic Support for Robust, Distributed Programs". *ACM Trans. on Programming Languages and Systems 5*, 3 (July 1983), 381-404.

14. Martin, T. Real-Time Programming Language PEARL - Concept and Characteristics. Proc. COMPSAC, Chicago, 1978, pp. 301-306.

15. Marzullo, K. and Owicki, S. Maintaining the Time in a Distributed System. Proc. 2nd Symp. on Principles of Distributed Computing, 1983, pp. 295-305.

16. Mok, A.K. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment.* Ph.D. Th., MIT, May 1983. MIT/LCS/TR-297.

17. Mok, A.K. The Design of Real-Time Programming Systems Based on Process Models. Proc. Real-Time Systems Symposium, IEEE, Dec., 1984, pp. 5-17.

18. Paul, R.P. Communication Primitives for Robot Control Systems. Private communication.

19. Reghbati, H.K., Chow, F.F.L. and Hamacher, V.C. Some Implementation Results in Real-Time Operating Systems. Proc. Canadian Computer Conf, Edmonton, Alberta, May, 1978, pp. 124-128.

20. Schwan, K., Bihari, T., Weide, B.W., and Taulbee, G. GEM: Operating System Primitives for Robots and Real-Time Control Systems. Proc. Int. Conf. on Robotics and Automation, 1985, pp. 807-813.

21. Scott, M.L. A Framework for the Evaluation of High-Level Languages for Distributed Computing. #563, University of Wisconsin-Madison, Oct., 1984.

22. Shin, K.G. and Epstein, M.E. Communication Primitives for a Distributed Multi-Robot System. Proc. Int. Conf. on Robotics and Automantion, 1985, pp. 910-917.

23. Spector, A.Z. "Performing Remote Operations Efficiently on a Local Computer Network". *Comm. of the ACM 25*, 4 (April 1982), 246-260.

24. U.S. Department of Defense. *MILITARY STANDARD Ada Programming Language.* 1983.

25. Wirth, N. "Toward a Discipline of Real-Time Programming". *Comm. of the ACM 20*, 8 (Aug. 1977), 577-583.

26. Wirth, N.. *Programming in Modula-2.* Springer-Verlag", 1982.