Eds: J. Barnes, G. Fischer. The 1985 International Ada Conference. May, 1985. pp. 72-84.

SOME PROBLEMS IN DISTRIBUTING REAL-TIME ADA PROGRAMS ACROSS MACHINES[1]

Richard A. Volz[2]
Arch W. Naylor

Trevor N. Mudge
John H. Mayer

Abstract

The Ada Research Group of the Robotics Research Laboratory at The University of Michigan is currently developing a real-time distributed computing capability based upon the premises that real-time distributed *languages* provide the best approach to real-time distributed computing and, given the focus on the language level, that Ada offers an excellent candidate language. The first phase of the group's work was on analysis of real-time distributed computing. The second, and current, phase is the development of a pre-translator which translates an Ada program into *n* Ada programs, each being targeted for one of a group of processors and each having required communication support software automatically created and attached by the pre-translator. This paper describes the pre-translator being developed and a number of issues which have arisen with regard to the distributed execution of a single Ada program, including language semantics, objects of distribution and their mutual access, network timing, and execution environments.

1. Introduction

"Ada" is the result of a collective effort to design a common language for programming large scale and real-time systems." So states the foreword to the Ada Language Reference Manual [DoD83]. This statement has often been elaborated to mean that Ada is intended for large, embedded, real-time systems executing in a coordinated fashion on a number of machines. Yet, to date, while tremendous effort has gone into the design of the language, the development of compilers for it, and the development of the Ada Programming System Environment, relatively little emphasis has been placed on the distributed and real-time issues. This paper addresses there latter two issues through the vehicle of distributed language, that is, one in which a single program may be executed on a distributed set of processors.

There are, nevertheless, a number of advantages to the use of a real-time distributed language capability, including:

- Real-time distributed systems are typically large and complex, and, consequently, difficult for a programmer or programming team to mentally encompass. The conceptual advantages associated with viewing the system as one large, highly-structured, program in one language are enormous.
- Interprocessor communication has been found to be one of most difficult and time consuming aspects of building complex distributed systems [VMG84], [VoM84], [Car84]. If this could be made implicit, the programmer could be spared a great amount of onerous detail. Fortunately, this is usually possible because the compiler can "see" the entire program at one time.
- Modern software concepts such as data and program abstractions [Sha80], and compile time error checking intended for the language level can be applied over the entire system as opposed to just over each of several individual parts with no checking between them.

[1]This work is supported by Land Systems Division of General Dynamics under contract #DEY-600483

[2]The Ada Research Group of the Robotics Research Laboratory, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, Michigan 48109.

- Synchronization and timing is, on the one hand, more straightforward for the programmer, while, on the other, the tedious details, as in the case of communication and conversion, are suppressed.

Once the need for a real-time distributed language is accepted, there are three choices: create a new language, modify an existing language, or, if feasible, use an existing one. Ada is an excellent candidate for the latter approach for a number of reasons. The Ada concept was designed to provide modern software tools for programming large, complex systems, to be highly portable, to provide closely monitored standards, to have an excellent support environment, and to provide programming mechanisms for real-time systems. Moreover, it provides mechanisms, e.g. pragmas, which can be implementation defined and are suitable for managing the distribution of a program in situations where the distribution is possible, while remaining consistent with the Ada language definition, even when distribution is not possible.

One approach to the distributed execution of a single Ada program would be to write an entirely new compiler and run-time system to manage the translation, and it may eventually be shown that this is the correct approach. However, it is not clear that enough is yet known about the ramifications arising from distributed execution to make the large investment necessary for this approach worth while. Instead, our group is taking a simpler approach. An experimental pre-translator is being developed which will translate a single Ada program into a set of inter-communicating Ada source programs, one for each node of the target network. Each of the Ada source programs created: (1) realizes part of the original Ada source program (typically this is close to a copy of a portion of the source); and (2) adds Ada packages to support the harmonious distributed execution of the resultant Ada programs. Each object Ada program is subsequently compiled by an existing Ada compiler for the processor for which the program is targeted, as illustrated in Figure 1.

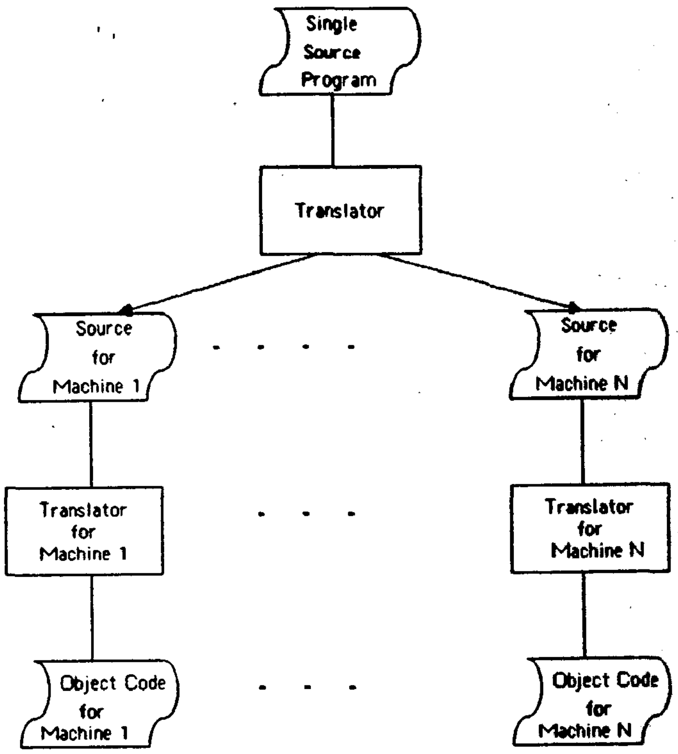


Figure 1

The development of the pre-translator is intended not only to provide an experimental tool for exploring many aspects of distributed real-time systems, but to expose language and implementation difficulties as well [VMN84]. Work to date has, indeed, revealed a number of problems in the distribution of Ada programs across heterogeneous processors. This paper discusses the more important part of these problems, organizing them under the headings of Definitional Issues, Object Access, Network Timing, and Execution Environments. This is followed by an introduction to the strategy being used to develop the pre-translator. Armitage and Chelini [ArC85] describe somewhat similar issues but in less detail.

2. Definitional Issues

Understanding the legal behavior of an Ada program which executes in a distributed manner requires extended study of the Language Reference Manual (LRM). Some issues which seem clear in the uniprocessor case are less so when distributed execution is considered. This section identifies some of these issues and discusses possible interpretations.

2.1. Objects of Distribution

The first question facing anyone who wishes to build a system allowing distributed execution of Ada is "What can be distributed?" The Language Reference Manual does not give an answer to this. Nor does it say how the distribution is to be specified. All that can be said is that the distributed execution of the program must be in accordance with the LRM. There are many levels of granularity at which one could define a set of entities to be distributed.

A rather coarse degree of granularity, which could be convenient from the perspective creating machine load units, is the use of packages as the objects of distribution. Through items declared in their visible part they can provide considerable flexibility in the items made available on remote machines. The distribution of most units smaller than packages creates a problem in building load units, as it becomes necessary to embed them within a library unit of some kind. For example, if a task or data item alone is to be distributed how is it to be stored and loaded on the remote machine? Tasks and data items alone can not be compilation units.

Nevertheless, in our experimental system we opted for a fine degree of granularity and allow the distribution of any object that can be created. Any object which can be allocated, data or execution, is allowed to be distributed. This choice was made for two reasons. First, it will allow us to explore the implementation strategies needed for all kinds of objects. Second, taking what are essentially the smallest meaningful distribution objects permits a study of distributed programming styles which is uninhibited by restrictive implementation decisions. The flexibility made possible by these two choices is important because systems that allow distributed execution are new and techniques for writing distributed programs (as opposed to writing collections of cooperating programs) have yet to be created.

2.2. Conditional Entry Calls

Conditional entry calls are a source of possible confusion in the distributed execution of a program due to network delays in calling across machines and the meaning of the word "immediate" in the semantic description of the call. The LRM states that "A conditional entry call issues an entry call that is then cancelled if a rendezvous is not immediately possible." The possible difficulty is in the word "immediate". At least one group [DGC83] has determined that due to network delays, conditional entry calls should always fail when the call is to a remote machine. However, the LRM also suggests a different interpretation when it restates the conditions for cancellation of the call, "The entry call is cancelled if the execution of the called task has not reached a point where it is ready to accept the call, .., or if there are prior queued entry calls for this entry".

If one adds the interpretation "when the call reaches the called task" to the second LRM statement given above, a clear interpretation results. This interpretation is independent of the time required to initiate the rendezvous. It depends only upon the readiness of the called task. This is appropriate. If a sense of time is required, timed entry calls should be used.

2.3. Timed Entry Calls

The timed entry call is the one place in the LRM where an upper bound on the time duration for some action to take place is stated. There are several questions to be considered with respect to timed entry calls. The LRM says both that the entry call ".... is cancelled if a rendezvous **is not started** within a given delay," and that if the "..rendezvous **can be started** within the specified duration ..., it is performed ...". (emphases added). The former implies that execution of the rendezvous must be started within the delay, while the latter implies only that it be able to be started within the given delay.

In most distributed situations the problem will be complicated, not only by a network delay, but also by an uncertainty in the consistency of the sense of time maintained on two or more processors (see section 4 for a detailed discussion of this point). Since there is likely to be an uncertainty in the difference in the sense of time available on two different processors, it may not be possible to make a precise determination of whether a rendezvous can or cannot be started within a given time interval. However, in many implementations it will be possible to provide bounds on the difference in the sense of time between two processors. This will make it possible to guarantee that if the rendezvous can be started within a calculable bound (as measured on the processor on which the called task resides), the called task can also be started within the given delay as measured on the processor from which the call was made. In these cases, there will be an uncertainty interval in which it will not be possible to determine whether or not the call can be started within the given time delay as measured on the processor from which the call is made.

An interpretation of timed entry calls for the distributed environment which would reflect these considerations would be that "if the call can be guaranteed to be able to start within the given delay it is started, and is cancelled otherwise". For the uncertainty interval, during which it might or might not be possible to start the called task, the timed entry call would be cancelled.

A second issues arises from the statement 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 arises with respect to whether the rendezvous is completed or cancelled. 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 Figure 2 below. To be consistent, there should be a single dividing line, above which the calls succeed (if the called task is ready) and below which they fail. A more consistent statement would result if the LRM did not contain the phrase about treating the case with zero or negative delay as conditional entry calls. Nevertheless, though unfortunate, the LRM does state quite clearly that the situation is as shown in Figure 2.

The implementation aspects of timed entry calls will be discussed further in conjunction with network timing in section 4.

3. Object Access

3.1. Modes of Access

The structure of Ada permits two different modes of access among execution objects (subprogram units or tasks). One is by passing parameters in subprogram calls or task entries. The other is by shared variables that exist in the common scope of the execution

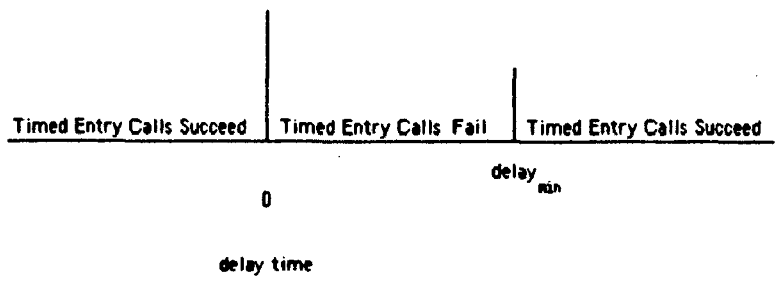


Figure 2 Time entry call success vs. delay time, assuming called task read to accept the call.

objects.

Ada requires that parameters be passed by copy. To avoid possible inefficiencies parameters that are arrays, records or task types may be passed by reference provided the effect is by copy. In the case of execution objects on tightly coupled machines[3] passing by reference, while keeping the appearance of by copy, can be efficient and makes sense. However, in the case of loosely coupled machines[4], passing by reference will lead to cross machine communication on each reference to the object passed. It thus seems natural to pass all parameters by copy. When the execution objects communicate over a Local Area Network (LAN), such communications are normally thought of as messages. This leads us to refer to access by copying parameters as message passing.

On the other hand communication between two execution objects through shared variables can be most naturally implemented with a shared logical memory. In the case where the shared logical memory is implemented as a shared physical memory this presents few problems. However, in the case where there is no underlying shared physical memory the run-time system must create the illusion. This leads us to refer to communication through shared variables as shared memory communication. If we return to the case of execution objects communicating over a LAN, but now consider shared variable access, the potential for inefficient communication becomes clear.

3.2. Addressing

The assumption that the objects of distribution may be any object that can be created in the language results in a large set of object access situations which must be explored. Objects may be created in three distinct ways: by declaration statement, by the **new** allocator, and, in the case of blocks, by their occurrence in the instruction stream. The types of objects which can be created by declaration statements are:

- scalars
- arrays
- records
- subprograms
- tasks
- packages
- access variables

The complications in object referencing arise primarily when one is implementing shared

[3] Machines that share physical memory.

[4] Machines that do not share physical memory.

memory access on loosely coupled machines. The Ada scoping rules make it convenient, though not necessarily well-advised, to write programs in this manner. Comparison with the tightly couple case will clarify this point.

3.2.1. Object Access in Tightly Coupled Machines

In the case of tightly coupled machines, shared data object references can be implemented as in a uniprocessor case. This requires that the underlying hardware memory protection system allow user processes on multiple machines to access the same regions of physical memory, but otherwise creates no problems for handling variables or pointers not already present in the language. Access to remote execution objects requires a signalling mechanism among the processors involved to permit the receipt of a remote call, but requires no special mechanisms for handling the actual parameters of the call. The tradeoffs between communication by shared variables or message passing are the same as for a uniprocessor implementation.

The principal difficulties that accrue to the translation system in the tightly coupled case lie in the recognition that a reference to a remote execution object has occurred. This recognition is straightforward if the distribution specification is done statically. However, if it is to be done dynamically, e.g. by placing a "site" pragma immediately preceding a new allocator for a task instantiation, an implicitly declared and assigned data structure is required to hold an identifier for the processor on which the execution object is to be placed. All references to execution objects via access variables must then reference this implicitly declared variable to determine the residency of the called object.

3.2.2. Object Access in Loosely Coupled Machines

In a loosely coupled architecture, the situation is considerably more involved, some of the solutions considerably less efficient and there are significant differences between shared variable and message passing communication. Each shared variable reference to a remote data object must be translated into a remote procedure call to a server of some kind on the processor holding the object. This server must then perform the required operation, and, if necessary, return a message containing the value of the object. On the other hand, if the variables are communicated via message passing references to them will be to the local copies and communication overhead will be substantially less.

The question of address representation is also immediately raised. The address of an object must include the address of the machine of residence. In the case of static distribution and direct references, this does not necessarily require any change to the local methods of address representation because the machine of residence can be identified in the pre-translator's symbol table and the pre-translator can place the code so that only local references are necessary, with messages sent between processors as needed.

Additional complications arise with either static or dynamic allocation when access variables are used because the pointer held in an access variable may reference an object on another processor, and the representation of that access variable might be different than the representation of access variables on the machine holding the object. The address of an object may be modeled, though not necessarily implemented, as a record with variant parts; the first component would contain a processor designation, and the variant part would contain the address of the object on the processor on which it resides. Note that this generalized view of addresses is required for pointer variables, though not for direct references, even in the static distribution case because assignments to access variables can change the machine containing the object referenced.

The referencing of remote execution objects requires, as in the shared memory situation, a signalling mechanism to permit the remote machine to receive the call. In this case, however, the actual parameters of the call must be sent to the remote processor, presumably via some type of message passing system. As noted above, for scalar variables the message passing is quite natural, since Ada requires call by copy. Although in the case of arrays and records, the LRM allows call by reference to be used under certain conditions, it would seems more appropriate to use call by copy since otherwise, each reference to the argument will involve the same kind of cross machine communication that occurs in the use of shared variables. The programmer always has the option of using access variables if it really is desired to access the arguments by reference. The problem can be further complicated by the fact that the actual arguments might not reside on the processor from which the call is made. In particular, if they should happen to reside on the same processor as the execution object being called, then in the case of records and arrays, the use of pointers might still be the most efficient method of parameter passing.

4. Network Timing

The Language Reference Manual does not absolutely require that an implementation provide delay timing; it is legal for an implementation to go away and never return on a delay statement. However, for many applications the language would be significantly reduced in utility without this capability. As the principal applications of interest here are real-time systems, all of the discussion in this section is pertinent to the situation in which an implementation does provide timing capabilities.

4.1. Network Sense of Time

The package CALENDAR provides functions which return values of type TIME. The implication is that there is a single sense of TIME throughout at least the execution of the program, if not between different executions of the program. That is, if CLOCK is called twice, with an intervening interval of one second, the calculated difference in the times should be one second. This poses an implementation problem when multiple processors are used for the execution of the program. How is a consistent sense of time maintained across the network? There are at least two possibilities, maintain a network time server to which all processors go when they need a value for time, or maintain separate but synchronized clocks on each processor. Combinations, of course, are also possible. Each has its own set of problems and limitations.

In the case of the network time server, the principal difficulty occurs because of the time required to access the time server. Two subproblems must be considered, the propagation delay, and interfering access requests. It might be possible to compensate for the first by subtracting the response time from the time returned, if the response time were reliably known. However, the second problem usually injects an uncertainty in the response time from the server. For some timer server configurations, however, it may be possible to bound the uncertainty in the time value returned.

The maintenance of perfectly synchronized local clocks is not possible. The best that can be done is to choose one as a master an update the others from it periodically. Between clock update points, there is an uncertainty of the difference between values of TIME read on different processors. The purpose of updating the clocks is to bound this uncertainty. One might, for example, try to keep this uncertainty less than one half of DURATION'SMALL. One experiment in maintaining synchronization among system clocks has been reported by Gusella and Zatti [GuZ84]. They found that to keep a network of VAX computers and SUN workstations synchronized to within 20 ms required updates once every 173 seconds. Scaling this to 25 microseconds (half of the 50 microsecond DURATION'SMALL recommended in the LRM) is moderately discouraging. Major improvements might be possible, though, by using a more stable clock in each of the processors.

4.2. Timed Entry Call Implementation

Most of the Ada constructs which reference time only require a local sense of time at each processor. For example, a delay statement within a task is simply a local delay with respect to processor on which the task resides. Similarly, the use of a delay alternative in a select statement with an accept statement is strictly local. There is one Ada construct, however, which if implemented in a nontrivial manner requires both an upper bound on the time within which a given action must take place (all other constructs just place lower bounds on time intervals) and a consistent sense of time among the distributed processors. This construct is the timed entry call.

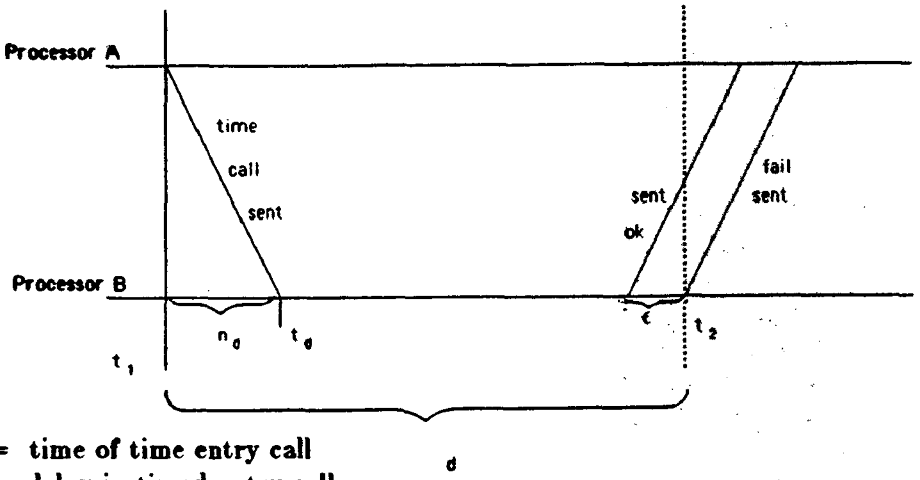
The trivial implementation of the timed entry call would be to say that there is no sense of time (across the network) and therefore that the rendezvous never takes place and the calling unit executes the alternative reference of code. If a nontrivial implementation is to be accomplished, then the timing of the action to be taken on the called unit must be determined with respect to the time scale of the calling unit. Otherwise, the language specification of the LRM cannot be guaranteed.

Consider a timed entry call made at time t_1, with a delay d, from a processor A to an entry on processor B. The time $t_2 = t_1 + d$ is the time by which the called task must be able to accept the call. Figure 3 illustrates the timing involved for non-negligible network delays. Two cases are shown. For case 1, the called entry is able to accept the call at 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 fails. Note that is both cases processor A cannot know whether or not the call was accepted until some time after t_2. This requires a liberal interpretation of paragraph 9.6 of the LRM which states that "..the entry call is canceled **when the specified duration is expired** and the optional sequence of statements of the delay alternative is executed" (emphasis added). Taking the alternative sequence (if present) at time $t_1 + d + n_d$ on processor A is consistent with the LRM if one takes the view that taking the alternative sequence only means making it ready at some time after t_2. The network delay, n_d, might or might not be known, or even bounded. It might well be different on the two transmissions. If $n_d > d$, then the rendezvous must fail.

4.2.1. Network Time Server

With a network time server, the scenario would be as follows:

- The processor containing the calling process will obtain the time from the network server and include both it and the specified delay in the timed entry call message



t_1 = time of time entry call
d = delay in timed entry call
n_d = network delay
$t_2 = t_1 + d$ = time by which called task must be able to accept.

Figure 3

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.

In this case the network delay used above must include the two timer server access times in addition to the call transmission time.

In order to obtain an expression for the local delay to be used on processor B in implementing this call, let t_1' be the value returned from the network timer server corresponding to time t_1 at which the call was made and t_a' be the value returned corresponding to t_a, the time at which the call message is received at processor B. Then, if the error in the times returned is bounded by d_B, then the local delay d_l satisfies the following inequality:

$$\begin{aligned} d_l &= d - (t_a - t_1) \\ &> d + t_1' - t_a' - 2d_B \end{aligned}$$

and the right had side of the inequality may be used to calculate the local delay on processor B.

4.2.2. Maintain Synchronism Among Local Clocks

An alternative method of providing timing is to maintain synchronism among the local clocks of the processors. Similar to the situation in the previous section, there will be an uncertainty interval in the difference between the measured values of the same instant of time between any pair of processors. For purposes of analysis, take the time measured on processor A as the reference and let t^A, t^B be the values for time t as measured on A and B respectively. Let $| t^A - t^B | \leq d_B$. Then the delay time from t_a to the upper bound for t_2 can be bounded as follows.

$$\begin{aligned} d_l &= d - (t_a^A - t_1^A) = d + t_1^A - t_a^A \\ &> d + t_1^A - t_a^B - d_B \end{aligned}$$

The RHS of the inequality can safely be used as a bound on the local delay time from the receipt of the request until the maximum value of t_2. Similarly, there is a minimum of delay time that can succeed.

$$d > n_d + d_B$$

4.2.3. Rely on the Exported Value of Delay

A third mechanism to manage timed entry calls is to export the time from the calling unit and use only this and local timing to manage things on the receiving processor. This requires knowledge, or at least a bound on the network transmission times. If $| n_d | \leq d_B$, then the receiving unit could use $d - d_B$ as a local bound on delay until t_2. The required existence of the bound d_B in the purest sense imposes limits on the type of network connection. Ethernets, for example, could not guarantee this bound; on the other hand, they might be acceptable in a practical sense.

4.2.4. Uniprocessor Considerations

Considerations such as those described above can be carried out in a uniprocessor situation as well. For example, the delay n_d corresponds to the overhead associated with implementing the checking and rendezvous. Indeed, these times should be included in n_d

in the distributed situation as well. Depending upon the granularity of delay interval implemented, n_d may be significant. This is likely to be the case for most processors at the 50 μsecond granularity recommended in the LRM and even more likely for the 10 μsecond granularity 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.

5. Execution Environment

Implementations of Ada are to provide several predefined packages as part of the environment available to the user. These include STANDARD, CALENDAR and TEXT_IO, and in general, must be available on more than one processor. The questions which arise are the consistency of data objects contained in or generated by subprograms in the package and the need to reference an object in one of these packages on a different processor, e.g. for I/O. These questions do not necessarily create a problem, but do require an awareness on the part of the programmer of the semantics associated with multiple occurrences of these packages.

In package STANDARD, the values for objects like SYSTEM.MIN_INT or SYSTEM.MAX_DIGITS may be different for the different occurrences of the package. Likewise, INTEGER, LONG_INTEGER, SHORT_INTEGER, etc. may have different meanings. The meanings, however, will be correct for the processor on which the package resides, and this is exactly what the programmer will need. As a matter of programming discipline, the programmer may find it useful to make greater use of some of the system descriptive objects to help in writing correct programs which can operate in the distributed environment. The distributed translator, however, must be aware of the possible differences in representation and supply the necessary translations. Also, it will be necessary to check values and, when necessary, raise exceptions, during the translation process.

Particularly in the case of I/O, it may be desired to reference an object supplied by TEXT_I/O from a processor other than the one on which the object resides. By embedding such requests in a block which is placed on the same processor as the referenced TEXT_I/O object, one can avoid the need to invent new naming conventions which might cause difficulties with the current definition of Ada.

Finally, since a fine degree of granularity is used, the implementation must provide a suitable shell (probably a package) to house distributed objects such as data items or tasks.

6. Experimental Translator Implementation

An Ada translator is being implemented which will convert a single Ada program into a set of inter-communicating Ada source programs, one to run on each node of the target network. The individual Ada programs will subsequently be compiled by existing Ada compilers, as illustrated in Figure 1. The mapping of objects to network nodes will be indicated by a pragma named SITE(.). When placed immediately before an object declaration, a new allocator or the occurrence of a block in the instruction stream this pragma will cause the following object to placed on the machine designator given as the parameter to the pragma. Any object created without a SITE pragma preceding it is assigned to the same node as the program unit in which the creation occurs. An alternative mapping scheme would use a distribution language which allows the same mapping information to be specified separately from the program itself as a sort of postscript [Cor84].

6.1. Translator Strategy

The global strategy for handling cross-machine references is based on the static construction of one or more special executable objects called agents. Each agent is designed to serve one particular executable object of the original program which makes an off-machine reference. The original executable object is called the master, to distinguish it from its agents. The agents will typically be of the same type as their masters. If during execution a master should need to access data or code located on a remote machine, it will order its agent on that machine to access the data or code for it. One master task may thus have several agents and in the extreme case, may need an agent on each of the other machines in the network.

To illustrate the general translation scheme, Figure 4 shows the source program and the translations of a distributed program for an autonomous vehicle. The example system has three interacting tasks: a planner, a vision system, and a drive control . The tasks are labeled PLANNER, CAMERA, and WHEELS, respectively. In the source program they are targeted for three different nodes. The translator will produce the three output programs shown. Note how PLANNER, itself residing on M3, has indirect access to both CAMERA through AGENT OF PLANNER ON M1, and WHEELS via AGENT OF PLANNER ON M2. For example, if the original PLANNER calls a procedure P within CAMERA, PLANNER will be modified so that instead making the reference directly, which is not possible, it will place the parameters of the call in a message which in then sends to its agent residing on M1. PLANNER's agent will receive the message, decode it, and discover that its master is attempting to call procedure P and, using the parameters that were included in the message, the agent will make the procedure call on its master's

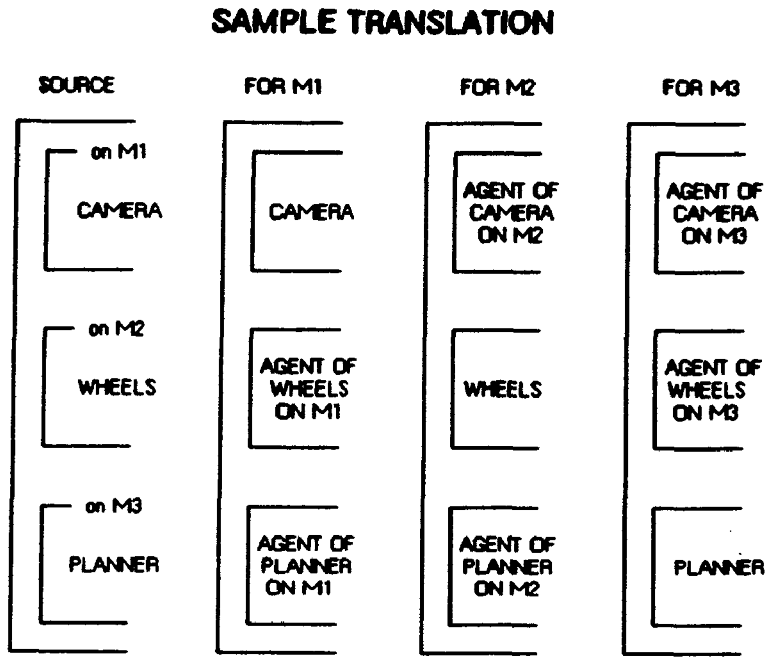


Figure 4 Distribution of source program to separate machine with agents inserted to represent task on remote machines.

behalf. After the call is done the agent will copy any out parameters into a message which is then sent back to the master PLANNER. The master copies these parameter values from the returned message into its local variables and, having completed the call to the remote procedure P, it continues execution.

The translator can be constructed in two distinct passes. The first pass produces an agent structure for each processor which is copy of the original program structure. Each executable object will have an agent on each processor which it (or an object it contains) references. The agents will be of the same type and will be nested in exactly the same manner as their masters, thus preserving the proper scope of objects created within them. This scheme, while generally applicable, can produce unnecessary messages among the processors (see further discussion below). The second pass is an optimization pass which removes these unnecessary messages.

More specifically, the first pass involves three basic operations, one on overall program structure, one on object declarations or blocks, and one on executable statements. The first operation forms the distributed structure of the output code, maps the input program into the appropriate parts and creates the necessary agents. It begins by extracting the skeleton of frames of executable objects, where the skeleton consists of the following parts:

- an opening line which marks the beginning of the frame, e.g. "declare" for a block or "procedure main is" for a procedure,
- the keyword "begin" which separates the declarative region of the frame from its executable code,
- the "end" statement which closes the frame.

These frame skeletons, reflect the nesting of the frames of the program. On the one node to which a frame is mapped, the skeleton encloses master version of the frame. On all others, an agent is created from the skeleton by adding an infinite loop which begins each iteration waiting for a command (message) from the master. After receiving a message, it executes a case statement which contains one choice for each of the remote operations required by the master. The agent uses the command to index into the case statement.

An agent is able to respond to any remote request it may receive from its master as long as both master and agent are in corresponding nesting levels in the program structure. To ensure that this correspondence exists, each agent will enter and exit its version of a frame in synchronism with its master. To illustrate, suppose that PLANNER is about to call procedure P, also located on M3. PLANNER's agents will have been executing server loops enclosed in their versions of the task PLANNER frame. Just before calling procedure P, PLANNER notifies its agents of the impending procedure call allowing them to switch frames as well. One of the advantages of this scheme is that any remote objects which have been declared in P will be allocated automatically as the agents enter their P frames.

The operation on data object declarations is fairly straightforward. An object resides only on the node to which it is mapped. There are multiple output streams, one for each machine in the network, and all streams are in synchronism with respect to the code being emitted. The declaration is simply placed in the skeleton of the agent for the machine on which the object is to be located.

The situation for remote object creation via allocators is slightly more complex, as it is both a run-time activity and involves pointers. The allocation expression is placed in the appropriate agent in a manner similar to the way declarations are handled, and the statement in original program is replaced by a remote procedure call to the agent, as described below. The pointer variable is placed in a record structure as described earlier.

Within an execution object, off-machine subprogram or task entry calls are replaced by remote subprogram calls to the appropriate agent which makes the call on behalf of the master and returns whatever results are required. Each reference to an off-machine data object, e.g. remote shared variables, is replaced by a remote subprogram call to the agent on the machine holding the referenced object, with an appropriate command code and any parameters required encoded into the call. If required, values are returned as function results, and used as normal in executing the statement in which the reference occurs.

7. Summary and Conclusions

A number of important issues which occur in the distributed execution of a single Ada program have been raised, and an experimental implementation of a translator which allows distributed execution described. The issues raised include the interpretation of the LRM in the context of distributed execution (e.g. constructs such as conditional and timed entry calls), the need for a consistent network view of time, and a number of implementation problems such as remote object access, network time management, data and address representations, and execution environments.

The experimental translator allows any data or named execution object to be distributed. It recognizes a pragma type named SITE as specifying the distribution. The translator takes a single Ada program as input and produces a set of Ada programs, one for each processor in the distributed computer network, as output. The general strategy for the implementation has been developed, and at the time of this writing, the translator is functional, but only partially complete, handling only simple distribution of tasks with no entry parameters.

References

[ArC85] Armitage, J.W. and J.V. Chelini, "Ada software on distributed targets: a survey of approaches," *Ada Letters*, vol. 4, no. 4, pp. 32-37, January-February 1985.

[Car84] Carlisle, B., "Sensor-based control: robot programming issues," *Workshop on Intelligent Robots: Achievements and Issues*, SRI International, Menlo Park, CA, Nov. 13-14, 1984.

[Cor84] Cornhill, D., "Partitioning Ada programs for execution on distributed systems," *1984 Computer Data Engrg. Conf.*, 1984.

[DGC83] Darpa, A., 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 of the European Communities, July 1982 - March 1983.

[DoD83] *Ada programming language (ANSI/MIL-STD-1815A)*. Washington, D.C. 20301: Ada Joint Program Office, Department of Defense, OUSD(R&D), Jan. 1983.

[GuZ84] Gusella, R. and S. Zatti, "TEMPO - A network time controller for a distributed Berkeley UNIX system," *Distributed Processing Technical Committee Newsletter*, informal publication of *IEEE Computer Society Committee on Computer Processing*, vol. 6, no. SI2-2, pp. 7-14, June 1984.

[Sha80] Shaw, M., "The impact of abstraction concerns on modern programming languages," *Proc. of the IEEE*, vol. 68, no. 9, pp. 1119-1130, Sept. 1980.

[VoM84] Volz, R.A. and T.N. Mudge, "Robots are (nothing more than) abstract data types," *Proc. of the Robotics Research Conference: The Next 5 Years and Beyond*, Aug. 14-16, 1984.

[VMG84] Volz, R.A. , T.N. Mudge and D.A. Gal, "Using Ada as a programming language for robot-based manufacturing cells," *IEEE Trans. on Systems, Man and Cybernetics*, December, 1984.

[VMM84] Volz, R.A., T.N. Mudge, A.W. and J.H. Mayer "Some Problems in Distributing Real-Time Ada Programs Across Heterogeneous Processers," *IEEE Workshop on Real-Time Operating Systems*, Wakefield, Mass., November 1984.