Proc. of the Society of Photo-optical Instrumentation Engineers Cambridge Symp. Advances in Intelligent Robotics Systems, SPIE vol. 727, Cambridge MA, Oct. 1986.

TRANSLATION AND EXECUTION OF DISTRIBUTED ADA PROGRAMS: IS IT STILL ADA?[†‡]

by

Richard A. Volz
Trevor N. Mudge
Gregory D. Buzzard
Padmanabhan Krishnan

Robotics Research Laboratory
College of Engineering
University of Michigan
Ann Arbor, Michigan 48109

Abstract

Intelligent control of Space Station will require the coordinated execution of computer programs across a substantial number of computing elements. It will be important to develop large subset of these programs in the form of single programs which execute in a distributed fashion across a number of processors. The single program approach to programming closely coordinated actions of multiple computers allows the advantages of language level software engineering developments, e.g., abstract data types, separate compilation of specifications and implementations, and extensive compile time error checking, to be fully realized across machine boundaries. As yet, however, there are few implementations of distributed execution systems.

Ada has been adopted for use in the Space Station, and the Ada Language Reference Manual indicates that distributed execution of Ada programs was in the minds of the language designers. However, when considered from the perspective of distributed execution, there are several aspects of the language definition which need further refinement, and a number of difficult trade-off decisions to be made in terms of translation strategies. This paper examines some of the fundamental issues and trade-offs for distributed execution systems for the Ada language.

1. Introduction

There has been considerable work done on the subject of parallel programming (see the excellent survey of [1]). The bulk of this work has concerned itself with shared memory architectures. In contrast, little has been done in the case of programs that run on distributed systems [2]. However, distributed execution of a single program is becoming increasingly important for embedded real-time systems as such systems are increasingly implemented with distributed microcomputers. The single program approach to programming closely coordinated actions of multiple computers allows the advantages of language level software engineering developments, (e.g., abstract data types, separate compilation of specifications and implementations, and extensive compile time error checking), to be fully realized across machine boundaries. As yet, however, there are few implementations which allow distributed execution of a single program.

While most efforts directed toward distributed programming have emphasized developing communication mechanisms and designing languages to accommodate distribution, we take the approach of adopting Ada and investigating its implications. We take this approach because Ada seems destined to become a major factor in embedded software systems, the Ada Language Reference Manual [3] indicates that distributed execution of Ada programs was in the minds of the language designers, and NASA has adopted Ada for the Space Station. This paper examines some of the fundamental issues and trade-offs for distributed execution of a single program written in the Ada language.

A few distributed Ada systems have been proposed and/or are in the process of being constructed. Cornhill [4, 5] describes the Ada Program Partitioning Language (APPL) for distributing an Ada program among a set of processors. This system permits the distribution of a wide variety of Ada elements. Jessop [6] advocates the use of a package type to allow programs in the language to dynamically create nodes. The extension to Ada implemented by Intel also includes a package type [7]. The package type, however, is a modification of the language. Armitage and Chelini [8] present a general description of four approaches to programming distributed systems in Ada. The approaches are described in general terms and no implementations or detailed designs are indicated. Indeed, Armitage and Chelini's fourth approach does not really qualify for distributed program execution.

[†]Ada is a registered trademark of the Department of Defense.

[‡]This work was partially sponsored by Land System Division of General Dynamics, Grant No. DEY- 601540 and NASA, Grant No. NAG 2-350.

The most comprehensive study to date is by Tedd, et al. [9]. They advocate an approach based upon virtual nodes. Full Ada is supported on each virtual node, which must support shared memory. Communication between virtual nodes is allowed only by task rendezvous. They describe an extensive system for constructing distributed programs at link time, i.e., the mapping of the programs onto processors is done after the program is written, providing greater flexibility in the construction of the execution system. However, it is necessary for the programmer to plan for the distribution by carefully designing the original program.

Mayer, et al., [10] describe some basic timing problems in cross processor task entry calls and describe a pretranslator approach which uses **pragmas** to specify the distribution. An important feature of this approach is that it can use existing compilers to perform the compilation. Based on the idea of [10], an Ada subset translation system for distributed execution has been implemented and is in operation at the University of Michigan.

Each of the above systems has either adopted a limited viewpoint or presented only a very general discussion lacking in detail. In this paper we examine some of the fundamental issues involved in translation for, and distributed execution of, Ada programs and the relation of these to the definition of the language. We conclude that in the context of distributed program execution several aspects of the language definition need refinement.

2. Preliminaries

Ada programs which are intended for distributed execution must deal with several forms of heterogeneity: heterogeneity of addressing program objects, heterogeneity of processing resources, and heterogeneity of the environment of the individual processors making up the distributed system. This section proposes that to account for this heterogeneity, a program definition must include some information on the distribution of the program. It further argues that the units of the language which may be distributed should be more precisely specified in the language definition. Finally, the major dimensions to the problem are identified and criteria which should be used in evaluating proposed translation/execution systems presented.

2.1. Distributed Ada Programs

Computer programs are written to produce output of some kind or have some effect on the environment. Embedded systems particularly emphasize the latter. However, programs do not, in and of themselves, have an effect; it is only their execution which produces an effect. When a program is executed on a uniprocessor, this distinction is generally unimportant and one often thinks of the program alone as producing the effect. However, when a program is executed in a distributed manner on a set of processors, the effect of the execution is impacted by an additional fundamental component, the mapping of the program onto the cooperating processors and memory. We will call the program/mapping pair an *execution object*.

It is thus the execution object which defines the effect which will result. For example, consider the control of a six degree of freedom robot by seven computers, one controlling each joint of the robot and one providing overall coordination of joint movement. Suppose that a task is assigned to the control of each joint. While the individual computers and interfaces may be identical, the effect of executing the program for two different mappings of tasks to processors will certainly be different; the robot would, in general, have drastically different motions. While the mapping details would certainly be hidden at higher levels of abstraction, it is also clear that the mapping must be explicit at some low level of abstraction as discussed in [11]. On the other hand, in many cases the effect of an execution object can be independent of the mapping component of the object.

It is also the case that translators whose outputs are intended for distributed execution must have some knowledge of the mapping. In general, the mapping can be static or dynamic, implicit or explicit, and come into existence and be used at any of several points in the program/compile/link/execute sequence. We divide the mapping into two parts. In the first part, elements of a program are designated as being distributable, without binding them to a specific machine, and certain characteristics of the mapping (roughly, the type of addressing required to access objects and the processor types which are to be able to execute fragments of code – see Sec. 4) specified. We call this part a *distribution specification*. The second part assigns elements of a program to specific machines. We call this the *binding specification*. The mapping is thus the pair (distribution specification, binding specification).

We will then define a *Distributed Ada program* to be an Ada program together with its distribution specification, and the portions of the binding specification necessary to define the effect of executing the corresponding execution object. The distinction between an execution object and a distributed Ada program is thus the bindings which are unessential to describe the effect of the execution. We will call the combination of the translation system, the distribution and binding specification mechanisms and the run time system which supports the translation and execution of distributed Ada programs a *distributed Ada system*.

2.2. Units of Distribution

The choice of units of the language which are allowed to be distributed significantly impacts both the translation process required and the execution efficiency obtainable. The Ada Language Reference Manual (RM) takes a step toward making the definition of distributable units a part of the language definition, but is not entirely precise. It explicitly states that parallel tasks may be distributed, and further, that any "parts of the actions of a given task" may be distributed if the effect of the program can be guaranteed by the implementation to not be altered. The latter would clearly imply that individual statements and even expressions could be distributed (which is highly desirable for parallel processing of some opera-

tions). It would seem that subprograms could be distributed. However, internal data objects and packages are not themselves actions or parts of actions. One might infer, therefore, that they may not be distributed, though this is not explicitly forbidden. Library packages are not mentioned at all; since their distribution is not explicitly forbidden, it might be inferred that they may be distributed. On the other hand, since what the RM does say about units of distribution is to explicitly permit some distribution, it might be inferred that anything not mentioned may not be distributed. Clarification is needed.

It is clear that the RM does not require distribution of anything. Nor does it imply that because an implementation chooses to distribute one kind of unit it must also allow distribution of other distributable units. It is not stated whether or not it is required that an implementation which allows a unit to be distributed in some circumstances must do so in all circumstances. For example, is it permissible to limit the distribution of statements to non-recursive contexts? Similarly, there is no indication of whether or not an implementation can choose to restrict the language in some way to accomplish the distribution, e.g., disallowing data objects in the specification of packages which have tasks that are to be distributed.

The latter two possibilities seem inconsistent with the philosophy of language uniformity apparent in Ada. Indeed, there are two principles which we feel should underlie the choice of distributable units: 1) the definition should be fixed and not a function of the dimensions of the problem, and 2) language uniformity should be maintained.

In Section 3 we explore the implications of the units of distribution on translation difficulty, efficiency of code execution, language uniformity and distributed programming expressibility in order to provide more complete background for the decisions which must be made regarding the above issues.

2.3. Dimensions of Distribution

There are three major dimensions which parameterize a distributed Ada system and which will impact both the translation and execution phases of the system, but which are not part of the language specification. These, together with some of their typical values are:

- the memory interconnection architecture of the system upon which the distributed Ada programs are to execute,
  - shared memory systems
  - distributed memory systems
  - mixed shared & private memory systems
  - massively parallel systems
- the binding time of the distribution,
  - prior to compile time
  - between front end and back end compilation phases
  - at linking time
  - at run-time
- the degree of homogeneity of the processors involved.
  - identical processors and system configurations
  - identical processors and different configurations
  - different processors, but similar data representations
  - completely heterogeneous

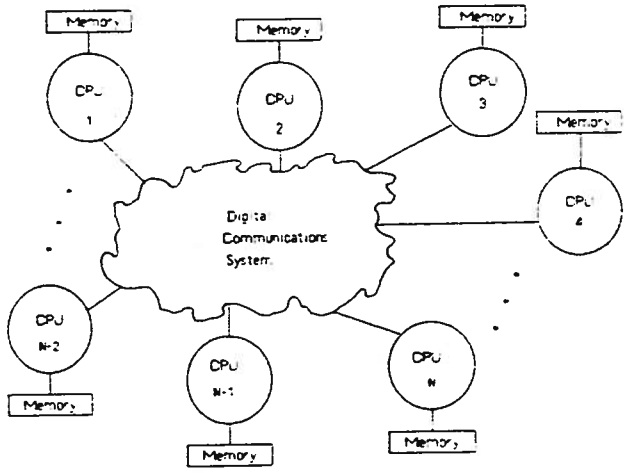

Figure 1

There are three major impacts of the memory architecture on the distributed translation system, the access time to objects, information which must be included in the distribution specification and the addressing strategies which must be used. Figures 1 and 2 illustrate two of the possible system architectures. Of particular interest is the mixed shared/private memory

scheme of Figure 2 since it both has a richer set of possible distribution modes requiring more complex implementation.

Only certain times for specifying the distribution and binding are reasonable, and depending upon the times chosen, several new utilities are needed for the compiler environment.

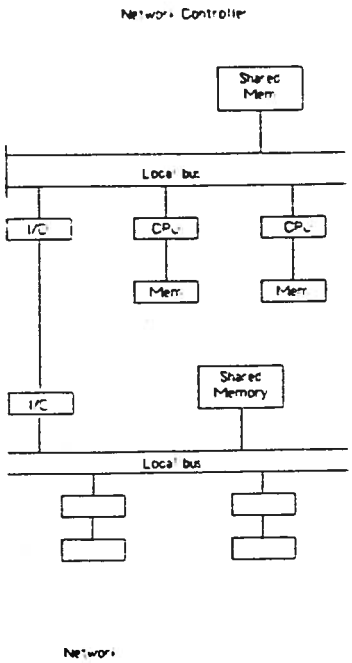

Figure 2

The impact of heterogeneity can be viewed in several different ways. First, it can be viewed as requiring translations between the data and code representations of the different processors. Second, it could be viewed as part of the semantics of the program. Or, the two views could be combined.

3. Unit of Distribution Considerations

We begin by examining the ways in which program elements can be assigned. There are three distinct kinds of location assignments to be made in the program mapping: 1) the memory unit to which data is assigned, 2) the memory unit to which code is assigned, and 3) the processor which is to execute the code. This classification is necessitated, in particular, by the mixed private/shared memory of Figure 2. Since each processor in this configuration has direct access to two memories, specifying a processor which is to execute code does not imply the memory to which either the data or code must be assigned. Similarly, since the shared memory can be accessed by multiple processors, assigning the code to shared memory does not imply which processor is to execute the code.

There are three types of addressing which will be called privately addressable (memory accessible only by the processor making the reference), shared addressable (shared memory) and remotely addressable (must be accessed via communication with another cpu). We will use the term directly addressable to mean that the addressing may be either shared or privately addressable. We require one rule of reasonableness, that the memory on which a code segment resides be directly addressable from the processor which is to execute the code. For most memory architectures this implies that the second and third cases collapse into one. It is only in the mixed private/shared case that the distinction must be made.

The comparison of units of distribution will be framed on four major issues that arise, in one form or another, for most of the possible choices for units of distribution. These are:

- Implied remote object access
- Object visibility and recursive execution
- Task termination problems
- Distributed types

The impact of the different choices for units of distribution on these issues will be discussed. Much of this analysis will be based upon interactions that are allowed among different elements of the language. It is important to note that all allowed interactions must be examined in considering the possible units of distribution, whether or not they correspond to good programming practice, since all interactions defined in the RM will have to be implemented.

An argument will be made that library subprograms and library packages are reasonable choices for the basic units of distribution. It will also be shown that to obtain reasonable execution speeds with this basic choice it will be necessary to distribute data objects corresponding to type definitions, and certain operations corresponding to these types.

3.1. Implied Distributed Object Access

Unless restricted in some way not currently specified in the language, the choice of packages, subprograms, tasks or blocks as units of distribution leads to a requirement that the code segment be able to reference remote data objects, subprograms, tasks and type definitions. This follows because in the cases cited in Sec.2.2 some executable object may be distributed from either the context in which it is defined or the context in which it is made visible via a **with**. Thus, either it must be able to reference the kinds of objects which can occur in the specification of that context, or, entities in its specification must be able to be referenced from that context. In particular, if a library package is a unit of distribution, then any subprogram or package including that package via a **with** must be able to reference any data objects, types, subprograms or tasks defined within it.

This implies a fine granularity of access, i.e., to individual data items. Except in the case of the mixed memory architectures, the time required for this access will involve both a communication channel delay and processing time on both processors involved. This delay will almost certainly be several orders of magnitude slower than accessing directly addressable objects, and will thus not be desirable for most applications.

There have, therefore, been suggestions that one avoid this delay by placing restrictions on what can be included in declarative regions or specifications to be distributed, e.g., disallowing data object or subprograms in the specification of a package to be distributed. There are two reasons why such restrictions are inadvisable. First, distributed access to data objects is highly desirable in some instances. For example, if one has a large database which is to be accessed in a number of different ways by tasks residing on different processors, a useful heuristic is to distribute the database in such a way that the individual data items reside in memory directly addressable by the processor which will most frequently operate on them. This implies a need for shared variables across machines. Even the distribution of small data objects makes sense in the context of a mixed private/shared memory. Second, such restrictions would be a change in, and disrupt the uniformity of, the language definition. One should not, for instance, allow packages in their full generality under some circumstances and disallow packages to contain data objects in others.

There is an important consequence of remote access to objects other than tasks with respect to translator implementation. Access to data objects or subprograms by code during its execution is part of the normal flow of control and normally given no special recognition with respect to the sharing of the processor, i.e., such accesses are not points at which the scheduler would normally be invoked. Since remote access involves sizable (in comparison to cpu instruction times) delay, remote references should be treated as points at which the scheduler is invoked so that other tasks may use the referencing cpu while the referencing thread of control awaits completion of the reference. Similarly, receipt of a message completing a remote reference should also be treated as a scheduling point.

3.2. Object Visibility and Recursive Execution

It is necessary to distinguish between the distribution of an object and remote access to it. As noted above, remote access to an object can be required as a consequence of distributing a larger item, such as a package. Distribution of an object itself means placing the object at a location different from the location containing the context surrounding the definition. While both imply a need for distributed access to the data object, the latter carries other implications as well. First, due to the possibility of recursive procedure calls, it implies the need for passing context information in some way with all references to the distributed object. Second, the implications of the program may be less clear to the programmer. We illustrate both points.

Suppose that the unit which creates an object (henceforth referred to as the C-unit), and the unit which refers to it (the R-unit) are at different sites. If the C-unit can be recursively called by the R-unit, many instances of it and its variables can co-exist. It then becomes necessary to export the context of the C-unit to all R-units accessing the objects in the C-unit to ensure that the correct version of the object is referenced. For example, consider the following pair of procedures involved in recursive calls:

```
procedure P1 is          -- Suppose this is the C-unit and is on machine M1
   X:INTEGER;
    |
   procedure P2 is       -- Suppose this is on machine M2 ≠ M1
     |
   begin
     |
     X := . . .          -- a remote reference
     P1;                 -- a recursive call
     |
   end P2;

begin -- P1
  |
  P2;
  |
end P1;
```

Since there will be many instances of the variable X, some mechanism must be developed to provide P2 with appropriate context information so that it can reference the correct instance of X, most likely by passing context information as an implicit parameter with the call to P2. In [10] P1 and P2 each have an agent on the opposite machine from which they reside, and communicate via a system of mailboxes. Each invocation of P1 instantiates a new version of P2's agent and creates a new mailbox through which P2 and its appropriate agent communicate. The mailbox id is passed to P2 upon its call, and essentially provides the proper context. This scheme has the advantage of being implementable with a pre-translator which allows existing Ada compilers to be used, but has the disadvantage of required an extra message to be passed at the exit of each call to P2 to tell its agent that it is done.

Similar problems of maintaining the proper context arise with the distribution of data objects, functions, tasks or blocks. This can result in a large number of messages between the sites and a corresponding loss of time if the C-unit and the R-units do not share a common memory. The cause of this difficulty is recursive subprogram calls in which some part of the recursive subprogram is remote from the rest. While it is generally inadvisable to write programs in such a way as to require this type of remote referencing within recursively called subprograms, if subprograms, tasks, blocks or data objects are themselves distributable (as opposed to being distributed as part of a coarser object such as a package), an implementation is obliged to implement mechanisms to allow such usage.

If only library subprograms and library packages are allowed as units of distribution, all instances of recursively created objects will reside at the same location as all units which reference them, with the possible exception of objects created via the **new** allocator. In the latter case, however, explicit address information is available and the problem will not arise. Thus, the use of library subprograms and library packages as units of distribution both simplifies translator implementation and eliminates one possibility for programmers to construct unnecessarily complex implicit inter-processor communication. In those situations, as indicated above, in which it is desired to distribute data objects, the objects to be distributed may be encapsulated into a package, and the package then distributed.

A further consideration in the distribution of data, subprogram and task objects is distributed programming expressibility. It has been frequently stated that it is the philosophy of Ada is to make explicit as much of the operation of a program as possible. Since remote access is much more time consuming than local access, it may, in some cases, be necessary to have control over the access time, i.e., to take alternative action if an access is not completed within a given time. Ada provides the timed entry call mechanism which can, in theory at least, be used for this purpose for task entry calls, although [10] discusses a number of problems in the implementation of distributed timed entry calls. However, there is nothing comparable for other forms of remote access, e.g., remote data or subprogram references. It would, therefore, seem to be desirable to at least make remote accesses explicit in a program so that the programmer or someone reading a program could easily distinguish remote and local accesses. With the distribution of data, subprogram or task objects, there is no such labeling mechanism available. Packages, however, must be explicitly imported into a program context, and if the **use** is not used, each reference to an object of the package must be preceded with the package name, flagging it as an external (to the present context) reference. To think of package names as possibly designating remoteness makes the interpretation of package names ambiguous and is far from an ideal solution. However, it can serve as a flag to the reader to check further. It is a weakness of Ada that an indicator of remoteness is not available in the language.

8.3. Task Termination

Ada task termination is dependent not only upon the task potentially terminating, but upon sibling and child tasks, and in some cases the parent task, as well. There are several ways in which this can cause termination difficulty when the tasks are located on different machines. Consider the following code fragment:

```
task body MASTER is
   task SLAVE_1 is
      entry ENTRY_1;
   end SLAVE_1;
   |
   task SLAVE_4 is
      entry ENTRY_1;
   end SLAVE_4;
   |
   task body SLAVE_1 is
   begin
      loop
         select
              accept ENTRY_1;
           or
             terminate;
         end select;
      end loop;
   end SLAVE_1;
   |
```

```
task body SLAVE_4 is
|
begin
   loop
      select
           accept ENTRY_1;
        or
          terminate;
      end select;
   end loop;
end SLAVE_4;
|
begin                  -- MASTER
|
end;
```

Suppose that MASTER has reached its **end** statement and completed. It will terminate if SLAVE_1 ... SLAVE_4 are all at their select statements and waiting on an open **terminate** alternative. In a uniprocessor situation, this does not cause unusual problems. The run time system can check SLAVE_1 ... SLAVE_4 for waiting at the **terminate** alternative. The key point is that because it can run at the highest priority it can do so without any other task gaining control and making an entry call to SLAVE_1 ... SLAVE_4 before it completes the check and takes appropriate action.

With distributed execution this is not always possible. Suppose that MASTER is on processor M0, SLAVE_1 on M1, and SLAVE_2 on M2, etc. Now, when MASTER completes, it must check termination conditions on the other processors. Due to propagation delays, race conditions can arise. For example, suppose that MASTER has completed and serially checks the status of each of its slaves and that the timing of the events is as shown in Fig. 3. In this figure, C indicates that the unit has completed. an X indicates that a task is waiting on a terminate alternative, and a 0 indicates that it is neither completed nor waiting on a terminate alternative. T1, ..., T4 are the times at which the MASTER is sent messages from SLAVE_1, ..., SLAVE_4, respectively, indicating their state at those times. Note that at time T1, MASTER has been sent a message indicating that SLAVE_1 is waiting at a terminate alternative. Between times T1 and T2, SLAVE_4, which was not waiting at a terminate alternative makes a remote entry call to SLAVE_1, removing it from the condition of waiting on a terminate alternative. At time T2, SLAVE_4 has entered a state where it is waiting on a terminate alternative. Thus, SLAVE_1 ... SLAVE_4 all report that they are waiting at an open **terminate** alternative. MASTER might then terminate when it should not.

Of course, this problem could be blocked by making the slaves wait for further entries until all termination checking was done, but if there were a long list of sibling tasks some of which were not ready to terminate, this could cause SLAVE_1 to unnecessarily delay its operation. This problem can be addressed by a more complex termination polling strategy. However, that solution is not the issue here; it is the need for a complex strategy that is of interest. It can both increase the translation difficulty and and impede the execution efficiency of a distributed program.

3.4. Distribution of Types

Distributed access to subprograms and tasks (as might result from distributing packages) implies the need to use remotely defined types, as both the specification of the subprogram or task and the referencing unit must have visibility of the types of the arguments used. The distribution of types is one of the more interesting aspects of distributing Ada programs as it forces a consideration of unusual implementation mechanisms.

There are three questions which must be considered when objects (data or task) are created by units remote from the location of the unit in which the type is defined:

- Where are declared objects of the type located: on the site of the object declaration or the site of the type declaration?
- Where are allocated objects of the type located: on the site of the object declaration, the site of the type declaration, the site of definition of the corresponding access type, or the site of the declaration of the corresponding access object?
- Where are the operations of the type located?

For example, let data type A be defined in a package residing on machine M2, and X an object of type A declared in a unit residing on machine M1. If X were placed on M2 every reference to X from the unit in which it was declared would require a remote reference. Thus, it is likely one would want X placed on M1. One must then examine the implications of the operations associated with type A. Each defined data type has three classes of operations, basic operations, implicit operations and user defined operations. Some of the basic and implicit operations clearly should reside on M1, e.g., addition on numeric types, storage allocation for objects of the type, etc. To maintain uniformity, then, all implicit and basic operations should be imported to the machine on which the declared object resides. This, in turn, implies that the basic and implicit operations of distributed types must be replicated on all processors containing units which use the types.

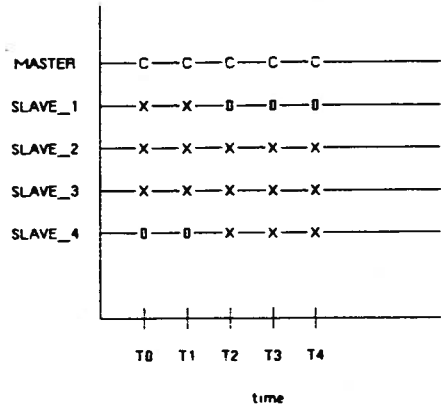

Figure 3

Applying the notion of language uniformity, then, one might expect that user defined operations should also be replicated on all processors containing units which use the types. However, user defined operations appear explicitly in the region in which the type is declared in the form of subprograms, and except for parameterless subprograms, all subprograms are operations for some type. Thus, replicating the user defined operations of types roughly equates, in packages for instance, to replicating all of the subprograms appearing in the package specification. This would also seem quite counter to what one would expect from distributing a package, which might after all only contain types and subprograms in its specification. Further, replicating user defined operations implies a remote access to variables and subprograms defined within a package body. It thus seems to the authors that it is only a slight sacrifice in language uniformity to not replicate user defined operations and keep them on the memory to which the unit defining the type is assigned.

Now consider object creation via the **new** allocator. This requires the definition of an access type for the object and the creation of an access object to hold address of the allocated object. Each of these could potentially be declared in separate packages distributed to different locations than either the one holding the original type definition or the one which will ultimately execute the allocator. For example,

```
package P1 is              -- on machine M1
   [task] type A is . . ;
     |
end P1;


with P1;
package P2 is              -- on machine M2
   type B is access P1.A;
     |
end P2;


with P2;
package P3 is              -- on machine M3
   C: B;              -- declare an access variable to objects of type A
     |
end P3;


with P1;
with P3;
procedure P4 is            -- on machine M4
  |
begin
   P3.C := new P1.A;       -- allocate a new variable object of type A
     |
end P4;
```

In this case a remote access is required on each reference to P3.C regardless of where the allocated object of type A is placed. The number of off-machine operations is minimized by placed the allocated object either on M3 or M4. To maintain language uniformity, then, one might elect to place the allocated object on M4.

Another set of considerations arise if A becomes a task type rather than a data type. Tasks can then be dynamically instantiated and the programmer may wish to control their placement on different processors as part of the algorithm being developed, e.g., via **pragmas**. Or, one might wish to reduce or eliminate the task termination problem described above. Both of these goals, however, have negative implications in terms of run-time efficiency, distributed program expressibility, and translational difficulties.

Eliminating the distributed task termination problem requires that tasks be placed on the same unit as their parents; then all of the checking of termination conditions will take place on a single processor and can use the existing mechanisms for doing so. Thus, declared tasks would be placed on the processor of the declaring unit while tasks created through evaluation of the allocator would be placed on the processor holding the unit in which the corresponding access type definition was elaborated. Any other choice allows task parentage to be remote from the task object itself and thus leads to the distributed termination problem. This would require placing an allocated object of type A in the above example (with A now a task type) on M2 since that is where the access type is elaborated.

However, if task objects are located coincidentally with their parents or at an arbitrary location assigned by the programmer, the code for task objects would have to be replicated as was considered above for user defined operations on types. The same difficulty of having to access local variables declared in package bodies would arise, which would then be

remote with respect to the task body. This has obvious execution efficiency degradations if tasks utilize shared variables which they might well do in a controlled way since in this case we are talking about shared variables hidden in the body of a package. Moreover, it will become very difficult for a programmer to recognize which references will be to remote variables.

Clearly, the translation also becomes more difficult. For example, consider separate compilation of package bodies which contain task bodies for distributed task types. Since package P2 and procedure P4 could be compiled before the body of P1, the replicated task bodies would be called for by normal compilation procedure before the body containing them would have been compiled. This could be handled by making the compilation process more involved and creating a record of units requiring the task bodies as they are compiled. Then when the package body for P1 is compiled, this record could be checked to determine other processors for which the task bodies must be compiled. Nevertheless, it would be one extra level of complication.

There is not a good solution which satisfies all problems. We have just outlined several problems with placing task objects anywhere other than at the location containing the task definition. Suppose, then, that to be consistent with previous comments about using packages as the unit of distribution we place task objects with the corresponding task type definition. This means that implementors will have to face the distributed task termination problem, and allocated and declared tasks will often be remote from the creating units, and thus involve remote task entry calls. If it is desired to place a task at any particular node then that task, or task type definition, must be encapsulated in a package. Consequently one could not have the tasks of the same type occurring at more than a single node. This is very constraining for some problems. To avoid it would require having package types, as suggested by Jessop [6].

3.5. Units of Distribution

As we have seen, there is no choice of distributable units within the current definition of Ada that is devoid of difficulties of one kind or another. Our preference is for library subprograms and library packages. They represent a reasonable granularity of distribution, they provide reasonable flexibility of distributed program structuring capabilities, they do not require cross machine dynamic scope management, and they present minimum difficulty to the compiler implementors. Data objects created from remotely defined types should be placed with the unit creating them, with implicit and basic operations being replicated. User defined operations should remain on the unit elaborating the corresponding type definition.

It would be our preference to restrict task objects created from task type definitions to the units on which the corresponding type definitions are elaborated, and to have package types added to the language specification. However, failing that, we believe it is necessary to allow task objects to be placed on the unit initiating their creation and living with the concomitant problems.

In view of the fact that some of the decisions concerning units of distribution have significant implications on the distributed language, we believe that the allowed units of distribution should be specifically identified in the RM more explicitly than at present.

4. Impact Of Translational Dimensions On Distributed Execution

4.1 Distribution and Binding Specifications

There are three issues to consider with regard to this dimension: 1) *when* the distribution and binding specifications are made, 2) *what* is specified at these times, and 3) the *representation* of the specifications. The first two of these issues are closely related to the fact that different addressing mechanisms are required for private and non-private memory references. Indeed, it is this fact that leads to the need for separating the program mapping into the two specifications. We will argue that the third issue is another shortcoming of Ada vis-a-vis distributed programs.

4.1.1. Run-time Specification of Distribution and Binding Both the movement of an existing object and the creation and location of a new objects are capabilities one might like to have. Deferring both the distribution and binding specifications until run-time means that the compiler will not even know whether or not object references are private or non-private. It will thus either have to use a generalized addressing mechanism (i.e., create a virtual target machine for all object accesses), or use a private memory addressing mechanism which will then have to be dynamically converted to a non-private addressing mode for the objects to be dynamically moved or created at a remote location. The use of a generalized mechanism throughout would make local addresses unacceptably expensive. The dynamic conversion of a private memory addressing mechanism at run-time is likely to require changing the instruction stream, an effort normally associated with compilation, i.e., something akin to dynamic recompilation (at least backend processing) would be required. This is likely to be complex and unacceptably slow.

If the distribution specification were given prior to backend processing by the compiler, the compiler would be able to use the right form of addressing and only the correct values would have to be inserted when binding is given at run-time. A change in the instruction stream would not be necessary. For movement of objects this is effectively a relinking operation for all references to the object being moved, while for dynamic allocation only a single address would have to be established.

The above is the principal argument for providing the distribution specification by compile-time. Subsequent sections on memory architectures and processor heterogeneity will describe more completely the information which must be included

in the distribution specification. Even with this, however, the overhead associated with moving an object may be substantial because of the relinking process. This suggests that only infrequently referenced objects such as whole programs be moved. Dynamic creation and deletion of objects, however, may be critical to some algorithms.

The mechanism for expressing the program mapping is an important issue. Unfortunately, the Ada language does not have a complete set of mechanisms by which run-time binding can be conveniently expressed. The **new** allocator provides a method of dynamically creating a data or task object, but has no corresponding mechanism for specifying target location. **Pragmas** could be defined to supply this information, but since **pragmas** are compile-time things, dynamic binding would require using a construct like **case** selection with each case being a distribution **pragma** and an allocator. This is rather awkward, especially for parallel processors with a large number of processors. Further, there is no mechanism for dynamically creating and binding packages, which we have argued above are the natural units of distribution. Finally, there are no mechanisms at all for specifying the movement of an object.

4.1.2. Distribution and Binding Specification at Link Time or Before Distribution and binding specification at link time faces the same complexities described for run-time, except that the overhead is incurred before run-time. Again stating the distribution specification by backend compile-time is essential in a pragmatic sense.

The remaining choices are to specify the distributions either between the frontend and backend compiler phases or prior to compilation. The former clearly allows more flexibility in terms of changing the assignment of distributable units without requiring full recompilation, while the latter permits a pre-translation scheme (described briefly in the next section) to be developed which can use existing compilers.

For distribution and binding specification at link time or before, language mechanisms for expressing the distribution are not required. Separate utilities may be used to interactively specify the distribution and binding, or to read a separate "program file" of specifications. However, as noted in section 3, from a point of view program expressibility it is desirable that the remoteness of objects be explicit. Also, the behavior of real-time embedded systems will depend upon the program mapping as well as the program. Thus, there must be an easy way for the programmer or software maintainer to read and correlate the program and the mapping. Having the mapping represented explicitly as part of the program would minimize the opportunity for a programmer to miscorrelate the two parts. Hence, either the mapping should be present in the program initially, e.g., via **pragmas**, or a decompilation tool is needed which can reproduce the original program with distribution and/or binding specifications inserted in the program text.

In summary, the distribution specification should be given by compile-time. It should either be included in the language or there should be a decompilation tool which will recreate the program with the distribution specification inserted in the code. Dynamic creation or movement of objects is rarely used in real-time programs because of the overhead involved. In this case, similar tools are needed for the binding specification, if binding is not included in the program. The Ada language does not have adequate mechanisms for expressing dynamic allocation and movement of objects in the distributed setting.

4.2. Implications of Memory Architecture

The principal effects of the memory architecture are on the nature of the addressing mechanisms and the time required to access remote objects (thus impacting the decisions on what to distribute). It was noted above that it is necessary for the distribution specification to be made by compile-time. If the system consists of only a single type of memory interconnection and this is known to the compilation system all that is required is designating the objects which may be remote from the code which references them.

However, if more than one memory architecture type may be present, the distribution specification must be strengthened to include the type of connection between processors and the memory holding objects they reference. This is necessary so that the compiler can generate the correct type of addressing mechanism. For example, consider a loosely coupled system in which each of the individual nodes consists of a mixed shared/private memory multi-processor system. The mechanism for addressing an object in a local shared memory will almost certainly be different than the one used for accessing data in private or remote memory. The compiler needs to know the kind of relationships which will be present in order to generate the correct instruction streams. Actual binding, which can occur later, will then be essentially a linking operation which merely supplies specific values for address references.

The distribution specification thus becomes a set of relations between pairs of objects in which the relations correspond to kinds of addressing required for the first object to reference the second. This is different from the binding specification which makes an absolute assignment to each object. In fact, the relations for the distribution specification can be deduced from the binding specification. However, separating the weaker distribution specification and making only the distribution specification available at compile time provides greater flexibility in distributed program development. It then becomes necessary to check for consistency of the distribution specification. Further, when binding is finally specified, it is necessary to check the consistency of the binding specification with the distribution specification. Thus, the separation of the program to processor mapping into distribution and binding specifications, while increasing flexibility, requires the development of additional support tools.

Finally, there is an additional interaction between the memory architecture and binding time considerations. A massively parallel system, such as the newly available hypercube architectures, is almost certainly going to be used differently

than a modest sized loosely coupled architecture. The latter is likely to be used for embedded real-time systems in which each of the loosely coupled systems is attached to a different device, all devices must work together in a coordinated fashion, and the binding specification is known *a priori*. While the former may also be part of an embedded real-time system, it is likely to have some special function in the system, such as image processing, in which the regularity of the parallel structure is to be exploited in some way. In this case, as a consequence of the homogeneity of the processors, the use to which the individual processors are assigned is likely to be determined at run-time. This implies a need for dynamic creation and distribution of objects in the program. As noted in the previous section, Ada lacks adequate mechanisms to deal with this dynamic distribution.

4.3. Impact of Heterogeneity

Heterogeneity impacts both the semantics of a program and the mechanisms for translating and executing it. One obvious way to deal with differences in processor types is by designing programs for a virtual Ada machine and then making the compilers for each real machine produce code which effectively implements the virtual machine underneath the translated user program. However, there is often a significant loss of efficiency with this approach. From a more pragmatic point of view, it would be very advantageous if compilers produced for uniprocessor operation could be included in a distributed translation system with minimum modification. This would almost certainly require using whatever data representations and mechanisms were natural for a given processor type. It would also deny translation within the compiler for a virtual Ada machine. From the user's point of view it will thus be the combination of the processor type and the compiler that are important, and when we speak of a "processor type," we will actually mean the combination of the processor type and the translator for it. With this assumption, we reach the conclusion that a distributed Ada program must include processor type information in the distribution specification. Otherwise, the semantics of the program are ambiguous.

Consider, first, the representation of primative data types, e.g., integers and floating point numbers. Ada provides mechanisms to support portability which are useful for distributed execution as well. One can define data types in terms of the ranges needed and let the implementation choose the underlying base type from which the new type is derived, with errors being flagged if any processor cannot support the required range. However, programmers are not obligated to use these mechanisms and the translation system must then provide some type of data translation. Unfortunately, there is then no guarantee that a translation is possible, e.g., you can't represent a 64 bit integer from machine A with the 16 bits that might be available on machine B. Additional checking of the distributed program is necessary to ensure the compatibility of representation of data objects. Thus, knowledge of processor type is required as input to a distributed translation system.

Second, different processors may well have different values for implementation dependent constants such as SYSTEM.TICK, and may use different scheduling disciplines. These differences may all be in accord with the RM, but when a program intended for execution on a single processor is moved amongst different processors, drastically different performance may result. It is in general understood that the effect of the program is dependent upon the implementation. However, when a distributed program is redistributed amongst a set of processors, the underlying implementation remains the same (even though the performance might well change), and it is no longer appropriate to think of the effect of the program depending upon the implementation. In this case, we think of the semantics of the distributed program as changing with the mapping of the original Ada program onto the specific set of processors in the system. The programmer should thus know or be able to control the type of processor to which things will be assigned. The processor type information should be included in the distribution specification.

5. Conclusions

The distributed execution of Ada programs requires further consideration of two issues: the units which may be distributed and the specification of the program mapping onto the set of processors and memory to be used. It was argued that the units of distribution should be stated more precisely than presently done in the RM. It was stated that the program mapping may be divided into two parts, a distribution specification and a binding specification; the former should be a required part of a "distributed Ada program."

It was recommended that the natural units of distribution for Ada are library packages and library subprograms. These, in turn, require remote access to individual data objects, subprograms and tasks. Use of remotely defined types requires replication of implicit and basic operations at each site creating objects of the type. Dynamically elaborated objects, e.g., tasks, need to be placed at the site of elaboration, which creates certain difficulties with respect to implied access to remote variables and task termination. The availability of a package type would alleviate some of these difficulties.

The distribution specification should specify the processor type and memory architecture used for each part of the program. The inclusion of processor type makes explicit program semantics which would otherwise be undetermined due to heterogeneity, while the memory architecture part of the specification allows the compiler to generate the correct kind of distributed object access code.

There are two principal areas where the authors feel that (hopefully minor) extensions are needed to Ada to handle the distributed execution situation. Since the package is the recommended distribution unit, mechanisms for dynamically instantiating packages and specifying the processor on which the new package is to be placed are needed. Syntax is needed by which remote references can be made explicit. In addition, several new tools are required: 1) a mechanism for expressing the distribution of a program, 2) a checker to ensure that the distribution specification is consistent with language rules, 3) a checker to ensure that a binding specification is consistent with the corresponding distribution specification, and 4) a decompilation tool which can insert the distribution specification into the rest of the program (if it was not there in the first place).

Most of the issues raised in this paper are closely related to the language definition. The authors believe that these issues should be considered in conjunction with the 1988 language definition review.

Acknowledgements: The authors would like to thank Charles Antonelli of the University of Michigan and Roger Racine of Draper Laboratories for their valuable constructive comments.

References

[1] G. R. Andrews and F. B. Schneider, "Concepts and notations for concurrent programming." *Computing Surveys*, vol. 15, no. 1, March 1983.

[2] D.M. Harland, "Towards a language for concurrent processes," *Software Practice and Experience*, vol. 15, no. 9, pp. 839-888, 1985.

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

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

[5] D. Cornhill, "A survivable distributed computing system for embedded application programs written in Ada," *Ada Letters*, Nov./Dec. 1983.

[6] W.H. Jessop, "Ada packages and distributed systems," *SIGPLAN Notices*, Feb/Mar 1982.

[7] Intel, *in Reference Manual for Intel 432 Extensions to Ada, 172283-001*. Santa Clara, CA: Intel, 1981.

[8] J.W. Armitage and J.V. Chelini, "Ada software on distributed targets: a survey of approaches," *ACM Ada Letters*, vol. IV, no. 4, pp. 32-37, Jan/Feb 1985.

[9] M. Tedd, S. Crespi-Reghizzi, and A. Natali, *Ada for multi-microprocessors*. Cambridge: Cambridge University Press, 1984.

[10] R.A. Volz, T.N. Mudge, A.W. Naylor, and J.H. Mayer, "Some problems in distributing real-time ada programs across machines," *Ada in use, Proc. of the 1985 Int'l Ada Conf.*, pp. 72-84, May 1985.

[11] R. A. Volz 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.