T-SE/15/3//25785

Translation and Execution of Distributed Ada Programs: Is It Still Ada?

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

Reprinted from
IEEE TRANSACTIONS ON SOFTWARE ENGINEERING
Vol. 15, No. 3, March 1989

Translation and Execution of Distributed Ada Programs: Is It Still Ada?

RICHARD A. VOLZ, SENIOR MEMBER, IEEE, TREVOR N. MUDGE, SENIOR MEMBER, IEEE, GREGORY D. BUZZARD, MEMBER, IEEE, AND PADMANABHAN KRISHNAN

Abstract—**Distributed execution of a single program is becoming increasingly important for embedded real-time systems. The single program approach to distributed programming allows the advantages of language level software engineering developments to be fully realized across machine boundaries. This paper examines some of the fundamental issues and tradeoffs involved in the translation and execution of programs written in the Ada language and intended for distributed execution. The memory access architecture, binding time and degree of system homogeneity are the three basic dimensions in terms of which target systems can be described. Library subprograms and library packages are identified as natural distributable units of the language. The program to process/memory mapping and the unit of the language to be distributed are the key issues in the distribution of Ada. The implications of various alternatives for these are analyzed.**

Index Terms—**Ada, distributed programming, programming languages, real-time systems.**

I. Introduction

THERE has been considerable work on the subject of parallel programming (see the excellent survey of [1]). The bulk of this work has dealt with shared memory architectures. In contrast, little has been done on programming for loosely coupled distributed systems [2]. However, distributed execution is becoming increasingly important for embedded real-time systems as such systems are increasingly implemented with distributed microcomputers. The single program approach to programming 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 was in the minds of the language designers, and there is growing interest in the use of Ada for distributed systems. This paper examines some of the fundamental issues for distributed execution 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 packages, which might then be placed on different 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 without details of four approaches to programming distributed systems in Ada.

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. 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 **pragma**s to specify the distribution. This approach allows existing compilers to perform the bulk of the compilation. Based on the idea of [10], a translation system for distributed execution has been implemented and is in operation.

Each of the above systems has either adopted a limited viewpoint or presented only a very general discussion lacking in detail. Many key issues have been overlooked. The definition of Ada with respect to distributed execution is not clear from the *Ada Language Reference Manual* (RM). As a result, implementors are developing their own guidelines which, explicitly or implicitly, cha[illegible]e the language definition. A typical example is disallowing shared variables in remotely accessible packages. But, this

Manuscript received November 26, 1986. This work was supported in part by Land System Division of General Dynamics under Grant DEY-601540 and by NASA under Grant NAG 2-350.

R. A. Volz is with the Department of Computer Science, Texas A&M University, College Station, TX 88743.

T. N. Mudge and P. Krishnan are with the Robotics Research Laboratory, University of Michigan, Ann Arbor, MI 48109.

G. D. Buzzard is with the Naval Postgraduate School, Monterey, CA 93943.

IEEE Log Number 8825785.

0098-5589/89/0300-0281$01.00 © 1989 IEEE

flies in the face of the "no superset, no subsets" philosophy of Ada.

This paper presents some preliminary thoughts on a number of issues regarding distributed execution of Ada programs. For each we take the view of a strict interpretation of the RM and show that in some cases this can lead to undesired consequences. Some of the undesired consequences can be avoided by making additional assumptions about the language, e.g., the restriction mentioned above. This leads to the question in the title: "Is it still Ada?". We believe that further changes will be necessary, and that the formal bodies controlling the definition of the language should recognize questions regarding distributed execution and clarify or extend the definition as necessary.

The next section introduces some preliminary terminology, introduces the principle dimensions to the problem and sets the stage for Sections III and IV. Section III discusses the consequences of allowing various units of the language to be distributed and Section IV discusses implications of mapping a program into a set of processors. Concluding remarks follow in Section V.

II. Preliminaries

This section introduces and discusses the notion of distribution of a program and identifies two principal areas that must be investigated: 1) the specification of the program/memory mapping and 2) the units of the language which may be distributed. It argues that the RM is insufficiently precise with respect to both of these issues; the majority of this paper is devoted to exploring these issues. Then, three principal dimensions that can be used to describe different distributed execution systems are identified. Whatever rules and language definitions are developed must apply across all systems that can be described using these dimensions. Finally, it presents three criteria by which alternative distributable units and mapping specification policies can be compared.

A. Distributed Ada Programs

The operating system literature has defined the activity that results from mapping a program onto a particular processor as a process. We shall extend this term and call the activity caused by the mapping of a *single* program onto a *set* of processors and memories a *distributed process*. Factors associated with the mapping and the relation of these to the program itself are the focus of discussion in this paper. More precisely, the mapping is a set of relations between program segments and memories (associated with the different processors). It can be specified statically or dynamically, implicitly or explicitly, and be made at any of several points in the program/compile/link/execute sequence. Moreover, it is not necessary to specify the complete mapping in one operation or at a single point in the sequence. It is possible to specify that a particular program segment may reside at a different location than the rest of the program (without specifying where) at one point in the program/compile/link/execute sequence, and specify the binding to a specific memory at a later point in the sequence. Separating the specification of the mapping into stages is somewhat analogous to using logical I/O channels and deferring the association with specific devices to run-time. It can have advantages in terms of fault tolerance and reconfiguration without requiring complete recompilation.

Translators whose compiled code is intended for distributed execution must have some knowledge of the mapping in order to generate the correct kind of addressing for objects in the program. And, some algorithm development depends upon partial knowledge of the distribution (e.g., which program segments are located remotely from other program segments). We thus divide the mapping into two parts. The first part is called a *distribution specification* and the second a *binding specification*. Loosely speaking, the distribution specification designates elements of a program as being distributable, without binding them to a specific machine. The second assigns elements of a program to specific machines. We call an Ada program together with its distribution specification a *distributed Ada program*. A more precise definition of these terms is left to Section IV.

B. Allowed Units of Distribution

The units of the language that may be distributed are closely related to the program/memory mapping, and are the other major area to be discussed in this paper. They significantly impact both the translation process required and the execution efficiency obtainable. The RM takes a step toward making the definition of distributable units a part of the language definition, but is not entirely precise. A more complete statement in the definition of the language is necessary.

The RM 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. There are three issues to be considered with respect to this statement: 1) what the "effect of the program" is, 2) what "guaranteed by the implementation" means, and 3) what this statement implies about the units of the language that may be distributed.

Effect of a Program: Unfortunately, it is not entirely clear even in the uniprocessor case what the effect of a program is. We have valid Ada programs which produce different results depending upon the memory location in which they are loaded. These programs involve reading the system clock and can return different values because of differences in code alignment with respect to 8 byte boundaries and the number of memory references necessary to fill the processor pipeline after a program jump.

For a second example, consider a system which can run at either of two clock rates and a program that must respond to external interrupts. Since fielding high priority interrupts can have side effects by causing different alternatives to be taken in timed entry calls in unrelated tasks,

the program can produce different results depending upon the clock rate at which the processor is run. At some level of detail the "effect" of a program can be influenced by such factors as system paging, or the instantaneous state of file system and its mapping on to disk segments. Obviously, including time effects complicates the issue immensely. But, Ada is intended for embedded real-time systems and has time related constructs in the language. The point is, that it can be extremely difficult to specify a set of conditions sufficient for uniquely defining the "effect of a program."

One might thus conclude that distribution of anything should be disallowed. This conclusion, however, would be more restrictive than that imposed on uniprocessor systems. It seems more appropriate to discuss distribution in terms of the semantic descriptions of the various Ada constructs as defined in the RM. That is, if one is considering placing a certain object or code segment on a processor that is different from the processor containing the surrounding definitions, or if a subprogram or package included via a **with** clause is to be located on a processor other than the including unit, all rules of scope, visibility, timing, and separate compilability must still be satisfied.

Guaranteed by the Implementation: There are two interpretations to the above phrase. First, it might mean that when an implementation can determine that the effect of the program is not altered by distributing some action of the program, it can automatically move the execution of the action to another processor. One might then be tempted to add the interpretation that the action can only be moved automatically, and not under user control. Second, it might mean that if an implementation can guarantee that the effect of the program is never altered by placing a particular unit of the language on a different processor it can give the programmer the ability to decide where the unit is to be placed.

Since the first interpretation would seem to be beyond the current state of optimization technology, we will explore what an implementation must do to accomplish the second interpretation for various possible units of distribution.

Implications for Distributable Units: Even if the effect of a program were uniquely defined, interpretation of the statement is still not clear. It would clearly imply that individual statements and even expressions could be distributed (which is highly desirable for parallel processing of some operations). 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, although 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. Nor does it imply that because an implementation chooses to distribute one kind of unit it must also allow distribution of other 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, e.g., is it permissible to limit the distribution of statements to nonrecursive contexts? Similarly, there is no indication of whether or not an implementation may restrict the language in some way to accomplish the distribution. The latter two possibilities seem inconsistent with the philosophy of language uniformity apparent in Ada.

In Section III we explore the implications of the units of distribution on translation difficulty, efficiency of code execution, language uniformity, and distributed programming expressibility.

C. 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. 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 and private memory systems
  - —massively parallel systems;
- the binding time of the distribution:
  - —prior to compile-time
  - —between frontend and backend 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.

Two of the major impacts of the memory architecture on the distributed translation system are the access time to objects, and the addressing stategies which must be used. Figs. 1 and 2 illustrate two of the possible system architectures. Of particular interest is the mixed shared/private memory scheme of Fig. 2 since it 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.

Heterogeneity raises the obvious issues of translations between the data and code representations of the different processors. However, it also raises substantial questions of data type definition, e.g., what is an INTEGER in a system with both 8- and 64-bit processors.

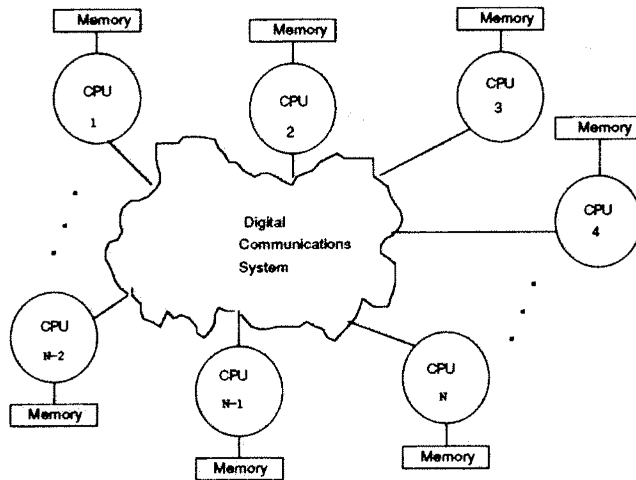


Fig. 1.

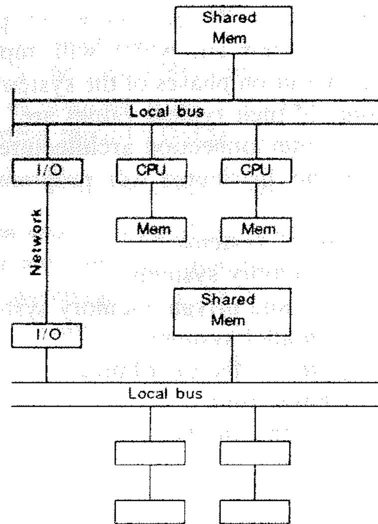


Fig. 2.

D. Criteria for Comparison

The comparison of alternative choices for units of distribution and distribution specifications is based upon one concept: access to remote objects. Based upon this, three different, and sometimes competing criteria arise, corresponding, respectively, to the programmer's, the translator's, and the run-time system's view of the distributed programming problem.

These are:

- distributed program expressibility
- translation difficulty
- execution efficiency.

Distributed program expressibility is concerned with the mechanisms for specifying the distribution of a program among a set of processors and memory. Are there external tools for expressing the distribution of the program? Or is the distribution expressible directly as part of the program? Should there be program notation to explicitly indicate remote objects, thus explicitly acknowledging the appreciably slower object access?

In the case of translation difficulty, the measure is the complexity of the constructs which must be included in the compiled code to ensure that access to remote objects can be accomplished while maintaining all of the other characteristics of Ada. For example, how much context information must be transmitted with remote object reference to allow correct address determinations to be made? For example, consider retaining Ada scoping rules with recursive procedure calls crossing machine boundaries. How are task terminations to be handled? How does one handle operations associated with remotely defined types? How is addressing of remote objects handled?

Execution efficiency, particularly for real-time operations, is perhaps the most important criteria. It is likely to be most influenced by addressing mechanisms for remote object references.

III. Unit of Distribution Considerations

A unit of distribution (or sometimes distributable unit) is a unit of the language which may be placed at any one of a set of memories. 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 Fig. 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 memory, 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 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 cases 2) and 3) above collapse into one. Only in the mixed private/shared case must the distinction 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 issues 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 be shown that it is necessary to place data objects declared from remotely defined types on the site of object declaration, and that certain operations corresponding to these types be replicated on the sites using them.

A. Implied Remote 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 programmer be able to reference remote data objects, subprograms, tasks, and type definitions. If a library package is a unit of distribution, then any subprogram or package including that package via a **with** clause must be able to reference any data objects, types, subprograms, or tasks defined within its specification.

This implies a fine granularity of access down to the level of 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 [9] 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 undesirable. First, remote access to data objects is highly desirable in some instances. For example, if one has a large database which is to be accessed 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 allow packages in their full generality under some circumstances and disallow packages to contain data objects in others. Rather, the generality should be allowed and the programmer allowed to access remote data objects if the cost of access is acceptable.

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 is 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 scheduling points. Similarly, receipt of a message completing a remote reference should also be treated as a scheduling point.

B. Object Visibility and Recursive Execution

It is necessary to distinguish between the *distribution of an object* and *remote access* to it. 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 with all references to a distributed object. Second, the implications of the program may be less clear to the programmer.

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, many instances of it and its variables can co-exist. It is then 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:

```
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 implicit context information 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 provides the proper context. This scheme has the advantage of being implementable with a pretranslator which allows existing Ada compilers to be used, but has the disadvantage of requiring 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 a way that requires 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. We call this the recursive context problem.

If only library subprograms and library packages are allowed as units of distribution, the recursive context problem does not occur, as no unit of code can be required to reference a recursively defined object residing on another processor. The Appendix gives a proof of this statement.

Thus, the use of library subprograms and library packages as units of distribution both simplifies translator implementation and eliminates one possibly for programmers to construct unnecessarily complex implicit interprocessor communication. Where it is desired to distribute finer grain objects, the objects 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 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, be used for this purpose for task entry calls, although [10], [12] discuss a number of problems in the implementation of distributed timed entry calls. 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 must be explicitly imported into a program context, and if the **use** clause is not used, each reference to an object of the package must be preceded with the package name, flagging it as an external reference. To think of package names as possibly designating remoteness makes the interpretation of package names ambiguous and is far from an ideal solution.

C. 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 without any other task gaining control and making an entry call to SLAVE_1 · · ·

SLAVE_4 before it completes the check because it can run at the highest priority.

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. 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 impede the execution efficiency of a distributed program.

D. Distribution of Types

Distributed access to subprograms and tasks 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

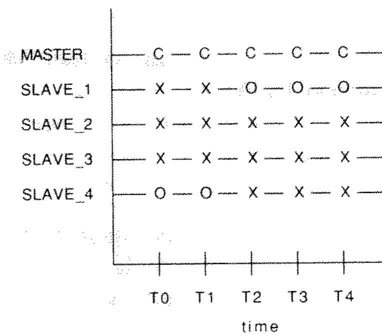


Fig. 3.

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 (i.e., replicated on) the machine on which the declared object resides.

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 to replicating all of the subprograms appearing in a 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 only 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 the 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: P2.B;                     —declare an access vari-
    |                            able 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 it should be placed on M4.

Additional considerations arise if A is a task type. Tasks can then be dynamically instantiated and the programmer may wish to control their placement as part of the algorithm being developed. 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. 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. Moreover, it will become very difficult for a programmer to recognize which references will be to remote variables.

To illustrate consider the example below:

```
Machine 1            Machine 2
with A;              package A is
procedure B is         task type T is
  T1: A.T;                 entry E1(· · ·);
    |                  end T;
begin                  T2: T;
    |                end A;
end;                 package body A is
                       X: · · ·;
                       task body T is
                         |
                       begin
                         |
                         X:= · · ·;
                         |
                       end T;
                     |
                     end A;
```

Note that T1 on machine 1 references the hidden variable X located on machine 2.

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 procedures before the body containing them would have been compiled. This could be handled by creating a record of units requiring the task bodies as such units 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. However, such a method is 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. To avoid this problem requires having package types as suggested by Jessop [6]. Along with the package types would come access variables to instances of packages. Further, access objects referencing package objects could be passed as in parameters to subprograms and task entry calls. Thus, task objects could be created with each package instance (on different machines) and access to these could be passed from one place to another via access pointers to the package containing them. Moreover, an extension to include package types could be up-

ward compatible with the existing definition of the language.

E. *Units of Distribution*

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, provide reasonable flexibility of distributed program structuring capabilities, do not require cross machine dynamic scope management, and present minimum difficulty to the compiler implementors. Data objects created from remote 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 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 live 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 they presently are.

IV. Distribution and Binding Specifications

A. *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. In general, one will need a distinct form of addressing for each type of memory that occurs in the system. In order for the compiler to generate code for efficient addressing, the distribution specification should include the type of connection between processors and the memory holding objects they reference. For example, consider a loosely coupled system in which each of the individual nodes consists of a mixed shared/private memory multiprocessor system. The mechanism for addressing an object in a shared memory will almost certainly be different than the one used for accessing data in private memory and this will be different from sending messages to access memory associated with a different processor. 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. The relations for the distribution specification can be deduced from the binding specification. 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 and distribution specifications. Thus, the separation of the program to processor mapping into distribution and binding specifications, while increasing flexibility, requires the development of additional support tools.

The memory architectures present in a system also have an impact on the general nature of the translation schemes that may be used. For example, the authors have implemented a pretranslation scheme which was intended to allow existing compilers to be used for distributed Ada programs [10]. Accordingly, only two addressing mechanisms were readily available, local object referencing and subprogram or task entry calls to remote objects. This suffices for distributed memory systems in which all object references are either local or require message passing between systems. However, in a mixed private/shared memory system, neither of these schemes will be satisfactory for the shared memory if the compiler's only knowledge of memory is the local memory attached to a single processor. The compiler would then be incapable of directly addressing the shared memory. Accessing it via system calls would be too inefficient. It would be necessary in this case that the compiler itself have knowledge of at least two kinds of memory, private and shared, and be able to generate efficient mechanisms for addressing both, defeating the purpose of the pretranslator approach. The pretranslator approach, however, can be quite useful for distributed memory systems with only moderate interprocessor communication speed requirements in which compile-time assignment of program elements to (logical) processors is acceptable.

Finally, there is an additional interaction between the memory architecture and binding time considerations. A massively parallel system 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, where the 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. Ada lacks adequate mechanisms to deal with this dynamic distribution.

B. *Binding Time*

There are three issues to consider with regard to binding time: 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 nonprivate memory references. We will argue that the third issue is a limitation of Ada vis-a-vis distributed programs.

1) Binding at Run-Time: Both the movement of an existing object from one memory to another and the creation and location of new objects are capabilities one might like to have. However, this requires deferring both the distribution and binding specifications until run-time since the type of memory access involved might change. This in turn means that the compiler will not even know whether or not object references are private or nonprivate. 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 nonprivate 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.

On the other hand, 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. It is also a restriction on the movement allowed; movement to a memory requiring a different kind of access could not be accommodated. 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.

Expressing the program to processor mapping is also an important issue. Ada does not have a sufficient 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 there is no corresponding mechanism for specifying target location. **Pragmas** could be defined to supply this information, but since **pragmas** are compile-time statements, 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 machines with a large number of processors. Since Ada does not have a concept of package types and generic instantiations are compile-time actions, 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. However, in view of the overhead involved in moving an object, this is probably not a serious limitation.

2) Binding at Link Time or Before: Binding 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 pretranslation scheme (described briefly in the next section) to be developed which can use existing compilers.

Language mechanisms for expressing the distribution are not required for distribution and binding specification at link time or before. Separate utilities may be used to interactively specify the distribution and binding, or to read a separate "program file" of specifications. However, from a program expressibility point of view, 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. Failing that, however, 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. The Ada language does not have adequate mechanisms for expressing dynamic allocation and movement of objects in the distributed setting.

C. Impact of Heterogeneity

Heterogeneity can be approached in several ways, but is far too complex to be fully addressed in this paper. Rather, we make some simple observations on three directions one might take, or possibly combine. One obvious method is to design programs for a virtual Ada machine and then make 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 unipro-

cessor 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 actualy mean the combination of the processor type and the translator for it.

Second, one can try to automatically perform appropriate translations on objects passed from one processor to another. With this assumption, we reach the conclusion that a distributed Ada program must include processor type information in the distribution specification. Consider, for example, the representation of primitive 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. Even in this case, however, the translation system may have to provide some type of data translation, e.g., 16 bit integers on one machine and 24 on another, or left-to-right byte packing on one machine and right-to-left on another. Moreover programmers are not obliged to use these mechanisms. Unfortunately, there is then no guarantee that a translation is possible, e.g., a LONG_INTEGER might be 64 bits on one machine and 32 on another. Either representation must be made more explicit or 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.

Third, one might consider processor type information as part of the semantics of the program. For example, 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 individually in accord with the RM, but when a program intended for execution on a single processor is distributed in different ways on a set of processors, drastically different performance may result. While it is in general understood that the effect of the program on the environment is dependent upon the implementation, 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 program units will be assigned, and this information should be included in the distribution specification.

In reality, aspects of all three approaches may be necessary. This is an area requiring considerable additional research.

V. 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 sepcification 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 could be upward compatible with the current definitions of Ada and 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 specification allows the compiler to generate the correct kind of distributed object access code.

There are two principal areas where the authors feel that 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. 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 Ada 9X language review.

Appendix
Theorem

If the units of distribution are library subprograms and library packages, the recursive context problem does not occur, that is, no unit of code can be required to reference

a recursively defined object residing on another processor.

Proof (by contradiction): In order for the recursive context problem to arise it is necessary that there be a recursively called creating unit, called the C-unit in Section III-B, which creates some object, call it X, which can be referenced by a code unit (an R-unit) on another machine. If the C-unit is a subprogram, then the R-unit must be definitionally nested with the C-unit. However this is not possible if the only distributable units are *library* subprograms and *library* packages.

Therefore, if the theorem is to be false, the C-unit must be a library package. In this case it is possible for another library subprogram or package (the R-unit) to reference an object declared within the C-unit (a library package). In order to generate the necessary recursion, it would then be necessary that the body of the C-unit include the R-unit via a **with.** In this case however the recursive call would involve a subprogram or task of the C-unit (package) and all dynamically created objects of that subprogram or task would be hidden from the R-unit. The only visible objects of the C-unit (package) are outside of the recursively called subprogram or task and thus are elaborated only once, during the initial C-unit elaboration.

Thus, while recursive calls can be constructed involving library subprograms and packages, the definitional nesting required to cause the recursive context problem cannot arise. Q.E.D.

Acknowledgment

The authors would like to thank C. Antonelli of the University of Michigan and R. Racine of Draper Laboratories for their valuable constructive comments.

References

[1] G. R. Andrews and F. B. Schneider, "Concepts and notations for concurrent programming," *Comput. Surveys*, vol. 15, no. 1, pp. 5–43, Mar. 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, Ada Joint Program Office, DoD, OUSD (R&D), Washington, DC, Jan. 1983.

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

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

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

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

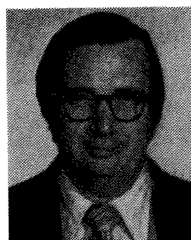
[8] J. W. Armitage and J. V. Chelini, "Ada software on distributed targets: A survey of approaches," *ACM Ada Lett.*, vol. IV, no. 4, pp. 32–37, Jan./Feb. 1985.

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

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

[11] R. A. Volz and T. N. Mudge, "Robots are (nothing more than) abstract data types," in *Proc. SME Conf. Robotics Research: The Next 5 Years and Beyond*, Aug. 14–16, 1984.

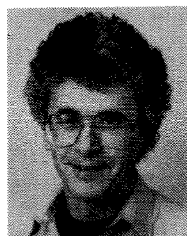
[12] —, "Timing issues in the distributed execution of Ada programs," in special issue on Parallel and Distributed Processing, *IEEE Trans. Comput. (Special Issue on Parallel and Distributed Processing)*, vol. C-36, no. 4, pp. 449–459, Apr. 1987.



Richard A. Volz (M'60–SM'86) received the Ph.D. degree from Northwestern University, Evanston, IL, in 1964.

He is currently Department Head of Computer Science at Texas A&M University, College Station. Prior to assuming his current position, he served as Associate Chairman of the Department of Electrical and Computer Engineering, Associate Director of the University Computing Center, and had over six years experience as Director of the Robotics Research Laboratory, all at the University of Michigan, Ann Arbor. He has worked on computational techniques for automatic control systems and done pioneering work on computer-aided design methods for control systems. Real-time distributed computing systems are a more recent technical interest of his, and he has led in the design and implementation of both a real-time operating system, a higher level language for real-time control and a distributed language translator. His current research interests include: 1) distributed computer systems (particularly distributed languages), 2) manufacturing software, 3) embedded real-time computer systems, and 4) robotics (particularly task planning, programming and telerobotics). Particular projects include (CAD) model driven vision systems, automatic determination of gripping points on objects (from CAD information), distributed systems integration languages for real-time control, and telerobotics.

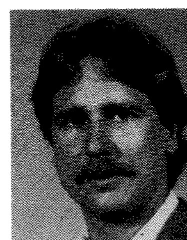
Prof. Volz served on the Automation and Robotics Panel of experts NASA established to give advice incorporating automation and robotics technology into the Space Station, and is currently a member of the Aerospace Safety Advisory Panel for NASA and Congress. He also has been active in systems integration and real-time computing systems, and is a member of the Ada Board, advisory board of the Department of Defense for policy on the programming language Ada.



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

He has been with the Department of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor, since 1977 and currently holds the rank of Associate Professor. His research interests are in computer architecture, programming languages, and computer vision, and he has authored or coauthored over 80 technical articles on these subjects.

Dr. Mudge is a member of the Association for Computing Machinery, the Institution of Electrical Engineers, and the British Computer Society.



Gregory D. Buzzard (S'80–M'82) was born in Detroit, MI, on April 27, 1959. He received the B.S. and M.S. degrees in electrical engineering and the Ph.D. degree in computer science and engineering from the University of Michigan, Ann Arbor, in 1981, 1982, and 1988, respectively.

From April of 1986 to August of 1988 Greg was also associated with the Center for Information Technology Integration (CITI) at the University of Michigan. In October, 1988, he joined the faculty of the Department of Computer Science at the Naval Postgraduate School, Monterey, CA, as an Assistant Professor. His research interests are in the area of architectures and operating systems for distributed and parallel computer systems.

Dr. Buzzard is a member of the IEEE Computer Society and the Association for Computing Machinery.

Padmanabhan Krishnan received the B. Tech. degree in computer science and engineering from the Indian Institute of Technology, Kanpur, India, in 1985 and the M.S. degree in computer science from the University of Michigan, Ann Arbor, in 1987.

He is currently a Ph.D. student at the University of Michigan, Ann Arbor. His research interests include programming language design and semantics, and real-time and distributed computation.

Mr. Krishnan is a member of the Association for Computing Machinery.