Distributed Ada: case study

R A. Volz, P Krishnan and R Theriault

I owe Prof. Volz a deep sense of gratitude for his valuable, generous guidance during my PhD and also during the initial phases of my career. He has this amazing willingness to help me in all aspects of my career. I have always valued his advice based on his deep insight fully knowing that at all times he has my best interests in mind at all times. He has always been open to my ideas and supported me in the decisions I made. His mentorship and encouragement enabled me to embark on my academic career. I wish him the very best of health and happiness and success in the years to come.

Paddy Krishnan

Distributed Ada: case study

R A Volz, P Krishnan and R Theriault

The paper describes the design and implementation of a distributed Ada system. Ada is not well defined with respect to distribution, and any implementation for distributed execution must make a number of decisions about the language. The objectives in the implementation described here are to remain as close to the current definition of Ada as possible, and to learn through experience what changes are necessary in future versions of the language. The approach taken to distributing a single program is to assign library units that compose it to nodes of the distributed system. In a formal sense the semantics of a program is independent of the distribution because the semantics is interpreted to include all possible behaviours that result from different distributions. However, the functionality of the distributed program may then depend on the distribution in the sense that program behaviour may be impacted by the time required for communication among the distributed modules, or parts of the program may continue to function in presence of failures. The implementation technique converts each distributed module into a standalone program that communicates with its correspondents; each of these may then be compiled by an existing Ada compiler. Issues discussed include the ramifications of sharing of data types, objects, subprograms, tasks, and task types. The implementation techniques used in the translator are described.

case study, distributed systems, Ada

The importance of distributed systems cannot be overemphasized, especially with the reduction in the cost of high-speed connection between powerful processing elements. Distributed computing has made inroads into many important areas, such as manufacturing, avionic systems, and space systems. The cost of developing software for such systems, however, is reaching astronomical proportions[1]. A major concern is the creation of software tools to harness economically the increased computing power.

Central to distributed software development is the language used to program these distributed devices. Distributed systems are still largely programmed by writing individual programs for each computer in the system, rather than programming the system as a whole using a distributed programming language. The single distributed program approach to programming closely coordinated actions of multiple computers allows the advantages of language-level software engineering developments, e.g., abstract data types (ADTs), separate compilation of specifications and implementations, extensive compile-time error checking, and large-scale program development support, to be fully realised across machine boundaries. This requires a single language capable of expressing distributed computation.

Ada[2] is one of the few languages that explicitly admits distributed execution. A principal shortcoming in the definition of Ada, however, is that it does not specify what parts of an Ada program may be distributed. The language definition just states that distribution must not change the effect of the program. The effect of a program, however, is not formally defined (such as with abstract operational semantics[3]) by the language designers. Thus the stance has to be adopted that the semantics of a program is actually a class of meanings or effects associated with the program. That is, the meaning is defined purely by the set of constructs used and is not dependent on the implementation (distributed or otherwise). A particular effect or behaviour, which is an element of this class of meanings, will be determined by the implementation. That is to say that the program is associated with a level of nondeterminism due to freedom in implementation. This is over and above the nondeterminism a program exhibits due to constructs in the language such as the **select** statement. More precisely, the 'distributed meaning' of a program is not one element; rather it is a class of meanings indexed by distribution similar to Gurevich's definition[4]. Given a program with a specified distribution, a given implementation will produce a given meaning (an element of the meaning associated with the distribution). Given this definition of distributed semantics, a system would be said to be correct if it produces an element in the class of meanings.

Due to the undefined nature of distributed execution, all of the implementations of distributed Ada place restrictions of one kind or another on what may be distributed. A number of experimental systems for distributing the execution of Ada programs has been developed[5-9]. As shall be seen, all these systems have been developed only after imposing various restrictions on Ada. A number of difficulties that these approaches must face if they are to remain within the current Ada definition has been described[10].

The goal of the work described in this paper is to understand the language and implementation issues that arise when distributed execution is considered. Therefore, the approach is taken that the language should be changed as little as possible, work done within its current definition, and then, when the study is complete, changes recommended, if necessary. A consequence of the above axiom is that the easy solution to problems that arise of 'lets change the language' is no longer avail-

Dept of Computer Science, Texas A&M University, College Station, TX 77843, USA

 0950-5849/91/040292-09

able. It means finding a way to implement a distribution of the language 'as it is'. This does not mean, however, that the language will not undergo change. If a solution results in a significant loss in efficiency or requires an overly contorted implementation mechanism, then changes to the language are recommended. But that is done after attempting to remain within the current definition.

The approach to studying the languages issues of distribution consisted of two phases. The first was to consider the problem in general and the second to choose an approach that 'minimized the number of problems' and actually construct a distributed Ada system. During the course of doing this, problems have been identified in both the language definition and translation rules and solutions developed. The actual construction is a proof that these solutions indeed work. The work described in this paper is principally an experimental device to help identify the basic problems/issues in general and point toward a solution for them.

GENERAL PROBLEM

The questions that must be faced in developing any distributed Ada are[10]:

- What units of the language may be distributed?
- How is the distribution specified (more recently called a partitioning activity)?
- How are the distributed units assigned to physical units in the system (called a configuring activity)?
- Is the system heterogeneous?

All these issues must be addressed by any distributed system. In other words, these problems are independent of a particular approach to distribution.

As the principal objective is to study the implications of distributing the current version of Ada, new restrictions, e.g., no shared variables, should not be imposed on the language if such restrictions can be avoided. The first question above, then, must be answered for the current definition of Ada. It has been shown[10] that for any reasonable choice for unit of distribution in Ada, a number of remote operations must be provided. These include:

- Declaring/allocating variables whose types are declared in remote packages.
- Reading and writing of data objects declared in remote units.
- Access to procedures and functions declared in remote units.
- Making entry calls on tasks declared in remote packages.
- Dynamically elaborating tasks whose types are declared in remote packages.
- Managing task termination for tasks elaborated across machine boundaries.

These problems require more than a standard remote procedure call[11] for solutions because of the presence of tasks, the remote visibility of types, and a variety of more subtle problems that arise when trying to implement a system for distributed execution. These issues are elaborated below.

Regarding partitioning and configuration, it is assumed that the programmer provides information about the logical distribution and subsequent mapping onto physical hardware. Once the principal question of the unit of distribution is answered, however, the strategy used must not impose any restriction on the nature of partitioning and configuration. For instance, if tasks are the unit of distribution, the programmer must be free to place any task at any logical location. Further research is necessary to support automatic partition and reconfiguration. Here consideration is limited to homogeneous systems. There are sufficient issues to study in a homogeneous system without the added complexity of heterogeneity.

Distributed types

The principal issues in allowing potentially remote units to share types are:

- Where are data objects declared from remote types located?
- Where are the operations on the type located?

These problems are over and above the problems described by Herlihy and Liskov[12], who discuss implementation issues about multiple representations. The problem introduced by a distributable language with separate compilation (such as Ada) is related to the fact that types can be in one module (hence onsite), the operations on it in another module (hence another site), while the object on which the operation is to be performed could be in a third module (hence third site).

The location of all objects of a type on the site where the type was declared is counter-intuitive. Normally, the location of an object would be identical to the location of the unit where the object is declared. If this is done, however, where are the operations of the type placed? If they remain only with the unit declaring the type, all operations would have to be remote, and that would be particularly awkward for basic and implicit operations, such as allocation, subfield identification, etc.

On the other hand, placing an object declared on the site holding the unit in which the type is declared also creates difficulties. For example, though an object is syntactically local to a procedure, any access to the object by the procedure, while appearing to be local, in reality would be remote. This is unacceptable, in general, but more specifically in real-time systems where it would be preferable for the performance to be identifiable from the syntax of the program.

Remote object accessing

The characteristics of data objects (in an imperative language in general, e.g., Ada) that cause difficulty in deve-

loping a general and yet efficient mechanism for handling references to remote objects are:

- The objects may be composite and may have concatenated names.
- Parts of a fully concatenated name may contain pointers that point to objects on other machines.

The first issue manifests itself when a composite object (as opposed to a simple object) must be copied from one site to another. For example, suppose that site 2 uses a record R on the right hand of an assignment statement and that R is located on site 1. The translated code must convert R to a bit string for transmission. It would usually be desirable that the part of the system that performs the conversion be unaware of the structure of the object. If the object contains a memory address as part of its structure (such schemes are useful when variable length arrays are used), however, the result received could be meaningless.

While some translation schemes might avoid this problem through careful handling of storage allocation, any scheme that relies on using existing compilers cannot. To overcome this problem, the routine that does the final message transmission must have knowledge of the structure so that the values themselves may be transmitted, and not just the address. Herlihy and Liskov[12] describe a scheme to overcome this by using a universal representation. They also describe the need for routines that deal with interchanging of formats between internal representation and the universal representation. A principal difference in their work and the problem in languages like Ada arises with objects involving operations such as '.' (the selector operation given a constructor) or '()' (the selector given a sequence or array). If these operators (along with routines that handle allocation/initialization, namely, the basic/implicit operators) are treated as part of an ADT definition, it would force the object to be created on the site where the type was declared. As described above, however, this is counter-intuitive. In short, while there is need to define a universal representation scheme and conversion procedures[12] for all types, it cannot handle all operations.

Consider the following example. Given a composite object (R) on site 1, let site 2 contain a statement such as X := R.C. How is an address for R.C constructed, or how is it described in a general way what element is to be returned? That is, how to send a message asking to receive the object R.C which is different from sending a message containing R and C to the operator '.' as with ADTs. The problem here is that the object is remote. The problem can be thought of as requiring a message containing '.' and C be sent to R. This is because the syntax 'R.C' exists only on site 2, and the only information available there from the specification of the package containing A is the logical record structure of R, not its physical structure, which is on site 1. Again, representation (by that unit) dependent knowledge of the rules used for construction of the universal structure of records is necessary.

If an access component, D, was now added to the type of R, and if the value of R.D was to point to another record stored on site 3, a second issue arises. The method to calculate the address of the item to be retrieved must not only contain implementation-dependent knowledge, but it must be distributed as well.

Object initialization issues

Another problem that arises is in the use of initialized objects in declaring types shared across sites. It appears obvious that sharing of types can be achieved by replicating the definition. A simple replication cannot be used, however, when the type definition uses an initialized object. This is because the function to initialize the object may cause side effects and replication could alter the semantics of the original program.

Anomalies in tasking

The problems discussed above are general as all imperative languages have the notion of objects whose values can be altered. That is, the above problems pertain to identifying an object and altering parts of it. The problem related to tasking, while specific to Ada, sheds light on how to define concurrency in languages.

Translation of distributed task types depends on the answer to the following question: Where is the placement of elaborated tasks: the site of elaboration or site of task type declaration?

The placement of tasks on the site of task type declaration is counter-intuitive and might render the programmer's load-balancing and fault-tolerant algorithms useless. On the other hand, if the task is placed on the site of elaboration another set of difficulties arise, which is discussed in the section 'Tasks'.

Termination

Any strategy to distribute task types (or any other structure involving task types) is not complete unless algorithms to detect termination of tasks created from such types are designed. Given the complexity of task termination, any strategy developed for task types should necessarily yield a simple termination algorithm.

To recap, issues have been identified that arise when a given Ada program is to be distributed. These problems are general and independent of approach. Certain problems may be eliminated by restricting the use of Ada, e.g., cannot share objects across machine boundaries. The posture is adopted here that the language should not be restricted on an *ad hoc* basis. Rather, all issues in the current definition should be addressed before suggesting changes.

OVERVIEW OF APPROACH

This section describes the authors' approach to address the general issues raised above. They have discussed the issues related to the memory architecture and units of distribution[13]. The main conclusion is that the choice of

the unit of distribution depends on the memory architecture. The notion of a virtual node is introduced to characterize a tightly coupled shared memory system. Tasks are recommended as the unit of distribution within a virtual node. As memory is shared among the processors, distribution is simply a matter of scheduling the tasks on the processors, as all objects within a virtual node are locally accessible.

A loosely coupled system can be considered to be a set of virtual nodes. Intra-virtual node access is local (as the memory is shared), while an inter-virtual node access is remote (as the memory is not shared). Volz[13] shows that the choice of Ada library elements as the unit of distribution is to a large extent concomitant with the notion of virtual node. At this time the implementation is restricted to homogeneous systems. Future design will address the issues related to heterogeneity.

The process of achieving a distribution of the program is divided into two parts. In the first, library packages and library subprograms are assigned to logical sites via a pragma SITE[10]. This is accomplished manually by the programmer. At this juncture manual specification is viewed as acceptable as the programmer has a better understanding of the coupling (such as inter-unit accesses) between the various units in the system and the system of implementation is independent of the specific distribution (as long as some distribution is specified). It is conceivable that a tool that computes the coupling and suggests a distribution scheme can be built. However, this is out of the scope of this research. The second aspect of distributed computation is that of binding of logical sites to physical sites, and this is done at the time of program load.

Other design decisions made about the more detailed issues raised above include:

- Data items whose type is defined in a remote package are to be located on the processor elaborating the object declaration. Basic and implicit operations are to be replicated on the processor elaborating the object declaration. This leads to a more efficient system as it obviates the need to send a message to select a particular component of a local object. User-defined operations, however, are not to be replicated.
- The only restriction to be placed on declarations allowed in the specification of a distributed package is that initializations from functions having side effects are disallowed. Shared variables across machines are thus allowed.
- Access to procedures and functions declared in remote units is permitted.
- Task objects elaborated from task types declared in a remote package are to be located on the processor creating the task. Hence task dependencies are either on library units, in which case the task is not required to terminate, or they are all on a single virtual node and the existing termination techniques can be used, i.e., there are no cross processor task dependencies involving tasks that are required to terminate.

Based on these design decisions, a translation strategy for translating a distributed Ada program has been under development and is nearly complete.

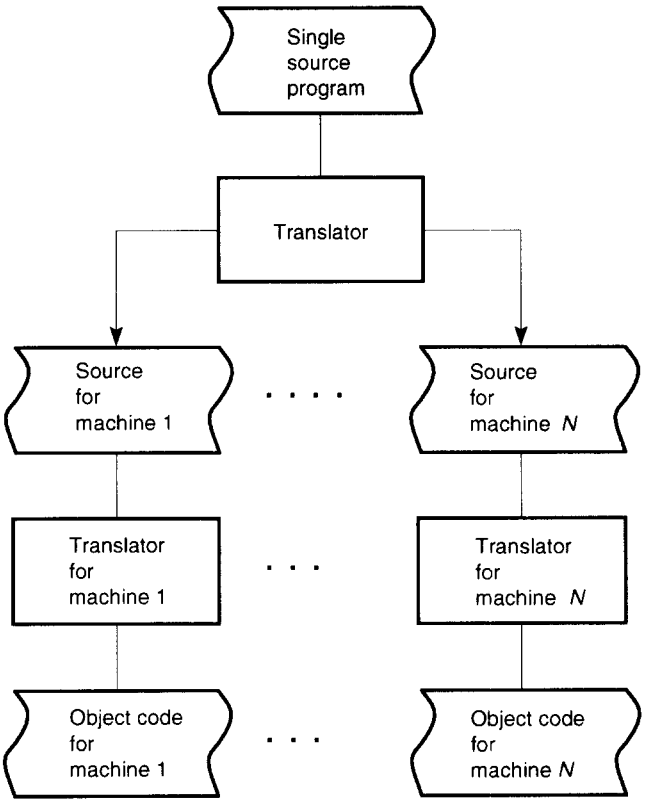

Figure 1. Translation of distributed Ada program into set of Ada programs

Overview of translation strategy

At the time the work was begun, the source code for an existing Ada compiler was not available for modification. Thus a decision was made to translate a single Ada program into a set of individual Ada programs that could each be compiled separately (see Figure 1). In the process, communication routines are added and references to remote objects or operations appropriately modified. This approach has the dual goals of being a simpler experimental mechanism and using existing work where possible.

As the type of remote services that can be requested from a package or subprogram are known from the specification, a well defined interface between units remotely requesting service and the package or subprogram providing the service can be created. The units that constitute the interface are called agents. A trio of agents are created for each library unit that may be accessed from remote sites; they are called remote agents, local agents, and common agents. Underlying these is a mail system called postal that is called by the agents whenever it is necessary to send information from one site to another.

For purposes of illustration, suppose there is a library package A that is to be accessible from remote sites. The remote agent for A is replicated on each site containing units that reference A. Suppose that a unit B makes

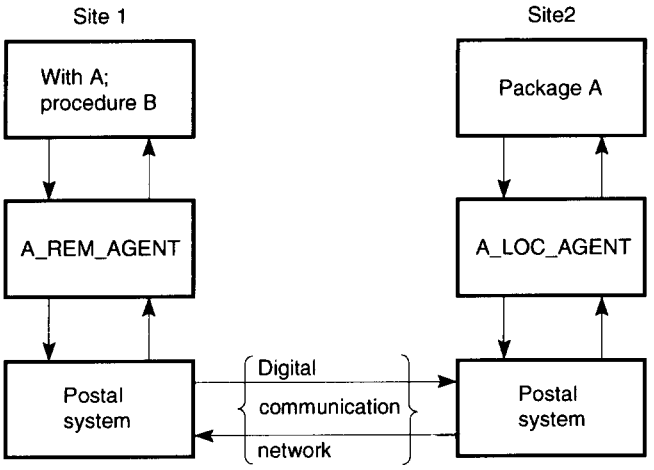

Figure 2. Use of local and remote agents to establish communication between packages assigned to different sites

references to A. The translator replaces all references in B to remote objects in A with appropriate calls to the remote agent of A (which resides on the same site as B), which in turn uses postal to convey the service request to the local agent of A. The local agent performs the necessary functions, returning any objects requested. A is essentially unchanged by the pretranslator. Both the local and remote agents can be generated only from the specification of A.

Common agents are placed on both sites and serve to propagate object accesses involving access variables pointing to other sites to the appropriate sites. Common agents are required only if such access variables are used. The organization of remote and local agents is shown in Figure 2.

Another key component to the translation strategy is a generalized mechanism for referencing objects, be they local or remote. The mechanism must be capable of dynamically processing a fully concatenated name, as the concatenated name may appear in a unit on one site and refer to an object on another.

TRANSLATION STRATEGY

As remote object referencing is used by some of the agents, the mechanism for dynamically handling fully concatenated names is described before discussing the agent structure.

Remote data object access

Central to the operation of the agent structure is the capability for handling reference to remote objects. An addressing scheme has been designed that uses the logical structure of the object rather than using physical structure constructed by the compiler, as it is more in keeping with the philosophy of using existing compilers where possible with minimal knowledge of their internals.

One of the principal problems in developing a general object accessing method is the processing of fully concatenated names (described above). The following constructs deal with this problem:

- An enumerated type, OBJ_ENUM_T, whose values are indicative of every data object declared in the package for which an agent is being generated.
- Enumerated types for all record types defined, which specify the field of the record.
- A collection of GETPUT procedures, one for each data, record, or array type defined, whose function is either to retrieve or to set the value of an object.

From the perspective of the local agent, a remote data object access begins with the local agent main task receiving a message from the postal system. One of the fields in this record contains a value of type OBJ_ENUM_T that indicates the name of the object being referenced. The local agent's main task then performs a case statement on this value. Each case calls a GETPUT procedure and passes it the message, the object named, and a count (DEPTH) of the number of name components to the fully concatenated name sought (including array arguments).

If the object passed is a scalar object, the count will be zero and the request can be satisfied directly by the GETPUT procedure by simply copying a value between the appropriate field in the message record and the object.

If DEPTH is not zero, then either an array element is being sought, or the final object designated by the fully concatenated name has not yet been reached. In the former case, the indices for the array element (or slice) are contained in the path array of the message record and the GETPUT can select the appropriate element(s) of the array. These either directly satisfy the request or recurse to the appropriate field in the composite object. That is, if the GETPUT is handling a record type, and DEPTH is not zero, the next element of the path array will contain an enumerated value that specifies the desired field of the record. The GETPUT contains a case statement conditioned on this field enumerator and an appropriate GETPUT is called, passing to it the message record and record field.

If one of the fields was an access variable, it would have been replaced by a record (as described in the common agent section below), and the action for the corresponding case would be identical to other field types. The GETPUT procedure for these access type records first checks to see if the requested object is on the current site or elsewhere. If local, then the call to GETPUT would be made as shown above. If elsewhere, then an appropriate message would be propagated to the common agent on the indicated site.

Agent structure

To motivate the discussion of the three kinds of agents, consider the following simple example:

```
pragma SITE(1);                pragma SITE(2);
```

```
package A is                    with A;
    type DT is . . . ;          package B is
    procedure P;                    ⋮
    task T is                   end;
      entry E;
    end;
end A;
```

One of the basic design decisions was that the data types and corresponding basic operations in A are to be replicated on each unit referencing A. To accomplish this, a package A_TYPES corresponding to A containing only the data types declared in A, namely, DT, is automatically generated.

A translation of B will among other things insert a **with** A_TYPES before B. Similarly, A_TYPES will be included with each package or subprogram that references A. B may also need to issue a remote procedure call to P or a remote entry call to T.E. Hence a remote agent for A to be used by B and a local agent associated with A that services the requests from B is automatically constructed. The agents can be considered to be generalized form of RPC stubs[14].

Remote agents

As B is remote with respect to A, B's references to A are altered to refer to A's remote agent, which is denoted A_REM. The form of A_REM is sufficiently simple that its specification can be generated from the specification of A.

Remote agents are collections of procedures and functions (P and E in the above example) that effect remote calls. In the case of subprogram calls, they present an interface to the calling package identical to that of the original source package. The procedures and functions in A_REM each format an appropriate message record and dispatch it to the appropriate site via the postal service. The value returned from the call is received from the local agent and returned to the calling unit. From the perspective of the calling unit, the facts that the action is remote and that there are (at least) two agents in between it and the called unit are transparent, except for the longer time required. Thus no translation of subprogram or (simple) task calls is required in the calling unit, unless they use arguments residing remotely.

In the case of remote data object references, a transparent interface is not possible. This is because while it is possible to use the same name for a subprogram/entry call but change the behaviour of the call, it is not possible to use the same object name to alter behaviour. For example, if x is an object, the use of x refers to the object and value returned is the value associated with the object. However, if x is a procedure x (i.e., a procedure call) can be used and the given body altered to take a different action than before. Thus the original program text (namely, x) need not be altered when x is a subprogram, but needs to be altered when x is a remote object.

Therefore, a set of procedures to access or update values of remote objects of various types is generated. Again these procedures (GETPUT described in the section 'Remote object accessing') are generated from the specification (of the object) of the library package being referenced.

Local agents

Local agents are packages consisting of three major components, a procedure called ROUTER, a queue manager task, and a number of call manager task instances. The main procedure created for each site contains a loop that first does a rendezvous with the underlying postal system to receive an inbound message, and then passes this message to the appropriate agent package by calling the ROUTER procedure within the agent package.

When the reference indicated by the message is to a data object, the ROUTER procedure takes action depending on the request, ensuring that a deadlock is not introduced due to its blocking.

```
procedure ROUTER(M: MESS_TYPE ) is
begin
  case M.OBJECT_ENUMERATOR is
    ⋮
    when OBJ_K_ENUM = > GETPUT_OBJ_1_
    TYPE(M, OBJ_1); return;
    ⋮
  end case;
  DEPOSIT_CALL(M);      -- message indicates a call
end;
```

When a message arrives requesting a call to a callable object, the message is placed on a queue managed by a queue manager task (QMGR). The local agent then verifies that there is an instance of a call agent task ready to rendezvous with the queue manager task. If there is not, one is instantiated. The call agent (described below) retrieves the queued message and executes the call.

A call agent task instance executes a call to one of the callable objects in the package and represents, locally, the thread of control on the remote processor for the duration of the call. The name of the actual object to be called is present in the message.

When a call agent has finished executing a call for a remote client, it does not terminate, but is queued up on the extract entry of the queue manager task and is thus able to execute subsequent calls. A local agent therefore defines one static task and instantiates N call agents, where N is the maximum number of simultaneously active calls to callable objects in the package, encountered during execution.

Common agents

The model of distribution permits access variables to point to dynamically created data and task objects on machines other than the one holding the access variable. Since access variables, as defined within a local Ada program, clearly cannot contain both the machine identity and an address, whenever an access type definition is encountered in the source package, it is replaced by a record structure containing two fields: a site number and the original access type. This new record type is then used in place of the access type. When a reference is made to an object via an access variable, the site number is always

checked against the current site number, to determine whether the object being pointed to is on the local site or on a remote site. If local, the object reference is handled as usual; otherwise, a call is made to a remote common agent to follow the pointer chain across processors to find the object and manage the operation.

Because access variables can be passed from one machine to another, it is possible for a processor to hold an access variable pointing to an object on a remote site. Moreover, the package that contains the remote object need not be otherwise referenced by any other package on the current site and may not even have a local agent. For these reasons, common agent packages are generated and placed on all sites that might use access types defined by the program unit.

Tasks

There are two difficulties to consider with tasks: remote conditional and timed entry calls, and creating task objects from a remote task type. The former requires a clarification in the wording of the RM and has been discussed[16]. It is not considered further here.

To illustrate the difficulties with creation of task objects from remote task types, consider the following package body skeleton, which defines the body of a task type.

```
with C;                    -- suppose a variable X for C is used.
package body A is
  S: SOME_TYPE;            -- a shared variable.
  task body T_TYPE is
    :
  begin
    — reference S;
    — reference X;
      :
  end T_TYPE;
    :
end A;
```

Recall now that when an instance of the task is created from this task body, the body must be replicated on the site on which the task object is to reside. Two problems are evident. First, the created task object must be able to reference the data object S. Second, the reference to X may be either local or remote, depending on whether the task object is created on the same site that holds C or a different one; the problem is that as package body A may be compiled before the unit that creates the task object, it is not possible to know whether the reference to X is local or remote at the time package body A is submitted for compilation. Of course, X may be local for some instances of the task and remote for others. The same is true of all other variables declared in packages references.

The first problem is one of the visibility of S. This is handled by pulling S (and its type declaration, if necessary) out of the body of A and creating a new package A_HID with S declared in its specification. Every package that creates an instance of the task from T_TYPE is then translated to include a **with** A_HID. While this does make internal variables visible, they are only visible to entities created by the translation process, not to the programmer. Thus this is not considered a violation of Ada visibility rules.

The second problem is handled by storing the code for task body T_TYPE in an auxiliary file. Whenever some other unit containing code that would create an instance of it is presented for compilation, an instance of T_TYPE is compiled for the specified site. Thus the number of compilations of T_TYPE is linear in the number of sites.

It is worth noting that, although the situation is not described further here, the same problem and solution arise with respect to compilation of generic units.

Automatic generation of agents, compilation, and visibility issues

The translations required for the methods outlined above involve numerous steps and introduce a number of auxiliary packages. The process of managing the auxiliary packages and compilation phases correctly is quite complex. Rather than require the user to do this manually, a utility has been prepared to simplify use of the pretranslator.

The translation and compilation procedure consists of the following steps:

(1) Determination of the order of pretranslation of source files*
(2) Pretranslation of source files
(3) Building main procedures and agents for all sites
(4) Determination of the order of compilation of translated sources (including agents) for target sites
(5) Compiling and linking of individual site programs

Several utilities have been written to facilitate some of these steps. A precompilation utility (ADAUTIL) translates the network of package dependencies implicit in a set of source files to a set of file dependencies in Unix 'makefile' format. The list of relevant source files and one or more targets (main programs) must be specified.

A second utility performs step 3. During step 3, all agent packages are constructed from saved symbol table information generated during the pretranslation process. Main programs for each site are also generated during step 3.

A third utility, (DAPLINK), builds a script to effect steps 4 and 5. If any non-Ada object modules need to be linked into any site, they may be specified by options to this utility. As a script (DAP) has also been written to effect steps 1–3, the user interface to the distributed compilation system reduces to two commands: DAP and DAPLINK. For example, let the current directory contain all and only the source files with names ending in

*Although the order in which sources must be pretranslated is the same as the order in which they would be compiled by an Ada compiler, all of the compilers used by the authors are inadequate in the area of determining this compilation order, when presented with a set of source files, especially if these source files are present in different directories.

'.a'. Let these be the complete set of source files comprising a distributed Ada program. Let sites 1, 2, and 3 be used and let MAIN be the name of the main program. The user only needs to execute DAP MAIN *.a and DAPLINK MAIN 1 2 3.

RELATION TO PAST WORK

The lessons learned during this research are primarily applicable to imperative languages. The general scheme is not directly relevant to functional or logic languages as their implementation techniques are vastly different. However, solutions to issues related to sharing of types may be used in such languages.

If a distributed system based on imperative principles (e.g., Emerald[17]) is considered, a more relevant comparison can be performed. The principal difference between the approach in Emerald and the authors' approach is the movement of objects. In Emerald (and Amber[18]), objects (including executable objects) can be moved from one site to another. In the authors' system, objects are not moved as the primary domain is loosely coupled systems. This is because Ada allows call by value-result (which is different from reference) without 'altering the behaviour' of the program. If Ada allowed only call by reference, every access to a remote object would have to be converted to a remote operation, which, needless to say, could be horrendously expensive. An alternative strategy is to ship the object to the site and thus perform local access. The issues related to moving objects have been discussed[17,18]. When dealing with simple objects a value can be sent to the remote site, all operations on it are performed locally, and the final result is sent back to be assigned to the object. However, this simple scheme will not work for executable objects such as tasks. The authors transform task objects to pointers to the object and ship the pointer. When a call using the pointer is made, the pointer is deferenced and a call is made to the site where the object was resident. The strategy is more efficient and was designed with real-time applications in mind. However, automatic load-balancing cannot be performed.

Other paradigms for distributed systems, such as Linda[19], do not fit the Ada model. Linda has the notion of tuple space and each process in the system writes/read into/from the tuple space. This creates a view of 'shared memory'. However, Ada has an asymmetric calling technique and there is no direct notion of tuple space. Thus the authors admit that they are unaware of the relevancy of their techniques to paradigms such as Linda. In summary, the authors' work is directly relevant to languages such as DP[20], Occam[21], etc.

As mentioned earlier, a number of experimental systems for distributing the execution of Ada programs has been developed. Tedd *et al.*[6] describe an approach that is based on clustering resources into tightly coupled nodes (shared bus) with digitial communications among the nodes. They then limit the language definition for inter-node operations (e.g., no shared variables on cross node references). Cornhill has introduced the notion of a separate partitioning language[7] that can be used to describe how a program is to be partitioned after the program is written. This language has been described in greater detail[22]. Again, neither of these approaches recognises the full problem space involved in the distributed execution of programs. A technique in distributed Ada based on treating packages and tasks as the unit of distribution has been described[8]. While the issue of remote entry calls is discussed, the other issues of sharing types, task types, etc. are not addressed. Diadem[23] describes a technique using remote entry calls to effect remote communication. However, it imposes various restrictions such as sharing of only types across sites, being unable to call subprograms, etc. Hence it is primarily the effecting of communication rather than distributing Ada.

STATUS AND CONCLUSIONS

At the present time, the distributed translation system is operational for distributed packages with remote access available to all items described here.

While the system is operational on a network of Sun computers, there is still work to be accomplished before the distribution of library packages and subprograms is complete. Although the strategy has been determined[16], work has not yet begun on handling timed/conditional task entry calls. The addition of exceptions alters the agent structures slightly as care has to be taken that tasks which terminate due to exceptions are recreated. This is because of the Ada tasking semantics, which states that a task is no longer active once an exception has been raised in it. The scheme to handle exceptions across sites is described elsewhere[24].

Based on the authors' experiences with building the experimental translation system, a number of conclusions can be drawn. The first is that a construct that can be replicated on various sites is required to allow types to be shared across machines. If there is no such construct explicitly within the language, one must effectively be created within the code produced by the compiler. This need is not satisfied by the concept of a package type as then state information such as mutable objects would also be replicated. Hence a stateless unit that can be replicated across sites is necessary.

Another issue that complicated the translation was the presence of hidden remote accesses in objects created from task types. Such task objects depend on the body of the unit encapsulating the task type. Hence it is not possible to generate versions from the specification alone. There was an increase in the complexity of a valid compilation order[25]. To avoid this problem, the creation of tasks from remote task types should be disallowed. In its place, a higher-level typing mechanism on a unit that encapsulates the task is required.

It has also been shown that it is possible to take a given program in language L and produce a set of programs for distributed execution in L itself. These can be tied into the networking software to achieve distributed execution. This enables existing software to be used for

single site programs. Of course, for this technique to work a sufficiently powerful language is needed.

So far only one point in the problem space has been addressed: homogeneous, loosely coupled systems with static distribution. Additional representation mechanisms are needed to describe limitations dependent on architectural considerations, to describe binding mechanisms, and to describe processor types (so that implicit data conversions can be accomplished). Moreover, it is necessary to require greater use of representational specifications on data types shared among multiple processors, particularly when those processors are heterogeneous. The authors are confident that other work[12,26] can be used with their approach as a solution for heterogeneous systems.

In conclusion, Ada is not adequately defined with respect to distributed execution. The construction of any system for distributed execution requires the implementation to make a number of decisions. However, as shown in this paper, with some modest language changes a reasonable distributed system can be built. Also note that the authors' research has concentrated on distributing a given program. The onus of configuration has been left to the programmer. To build a distributed system easily an associated configuration language should also be developed[27].

ACKNOWLEDGEMENTS

This work was supported by General Dynamics under contract no DEY-605028, General Motors under contract no GM/AES (1986–87), the Air Force under contract no F33615-85-C-5105, and NASA under contract no NCC 2-601.

REFERENCES

1 **Booch, G** *Software engineering with Ada* Benjamin/Cummings (1987)
2 **ANSI** 'Ada programming language' *ANSI/MIL-STD-1815A* Washington, DC, USA (January 1983)
3 **Plotkin, G D** 'An operational semantics for CSP' in **Bjorner, D (ed)** *Proc. IFIP TC2 Working Conf. Formal Description of Programming Concepts II* North Holland (1982) pp 199–225
4 **Gurevich, Y** 'Logic and the challenge of computer science' in **Borger, E (ed)** *Current trends in theoretical computer science* Computer Science Press (1987)
5 **Jessop, W H** 'Ada packages and distributed systems' *SIGPLAN Notices* (February 1982)
6 **Tedd, M, Crespi-Reghizzi, S and Natali, A** *Ada for multimicroprocessors* Cambridge University Press (1984)
7 **Cornhill, D** 'Partitioning Ada programs for execution on distributed systems' in *Proc. 1984 Computer Data Engineering Conf.* (1984)
8 **Bishop, J M, Adams, S R and Pritchard, D J** 'Distributing concurrent Ada programs by source translation' *Soft. Pract. Exper.* Vol 17 No 12 (December 1987) pp 859–884
9 **Hutcheon, A D and Wellings, A J** *Supporting Ada in a distributed environment* ACM SIGAda (May 1988)
10 **Volz, R, Mudge, T, Buzzard, G and Krishnan, P** 'Translation and execution of distributed Ada programs: is it still Ada?' *IEEE Trans. Soft.* (Special issue on Ada) (March 1989)
11 **Birrell, A D and Nelson, B J** 'Implementing remote procedure calls' *ACM Trans. Computer Syst.* Vol 2 No 4 (February 1981) pp 39–59
12 **Herlihy, M and Liskov, B** 'A value transmission method for abstract data types' *ACM Trans. Prog. Lang. Syst.* Vol 4 No 4 (October 1982) pp 527–551
13 **Volz, R** 'Virtual nodes and units of distribution for distributed Ada' in *Proc. 3rd Int. Workshop on Real Time Ada Issues* (June 1989)
14 **Jones, M B and Rashid, R F** 'Matchmaker: an interface specifications language for distributed processing' *ACM Trans. Computer Syst.* (1984) pp 225–235
15 **Birrell, A D and Nelson, B J** 'Implementing remote procedure calls' *ACM Trans. Computer Syst.* Vol 2 No 1 (February 1984) pp 39–59
16 **Volz, R A and Mudge, T N** 'Timing issues in the distributed execution of Ada programs' *IEEE Trans. Computer* Vol 36 No 4 (April 1987) pp 449–459
17 **Black, A P, Hutchinson, N C, Jul, E, Levy, H and Carter, L** 'Distribution and abstract types in Emerald' *IEEE Trans. Soft. Eng.* Vol 13 No 1 (January 1987) pp 65–76
18 **Chase, J S, Amador, F G, Lazowska, E D, Levy, H M and Littlefield, R J** 'The Amber system: parallel programming on a network of multiprocessors' in *Proc. 12th Symp. Operating System Principles* (1989) pp 147–158
19 **Gelernter, D** 'Generative communication in Linda' *ACM Trans. Prog. Lang. Syst.* Vol 7 No 1 (January 1985) pp 80–112
20 **Hansen, P B** 'Distributed processes: a concurrent programming concept' *Commun. ACM* Vol 21 No 11 (November 1978) pp 934–947
21 **Pountain, R** *A tutorial introduction to OCCAM programming* Inmos Corp. (1985)
22 **Honeywell Systems Research Center** *The Ada program partitioning language* Honeywell Systems Research Center, Minneapolis, MN, USA (September 1985)
23 **Atkinson, C and Goldsack, S J** 'Communication between Ada programs in DIADEM' in *Proc. 2nd Int. Workshop on Real-Time Ada Issues* (June 1988)
24 **Krishnan, P, Volz, R and Theriault, R** 'Distributed exceptions in Ada' in preparation (1990)
25 **Krishnan, P, Volz, R and Theriault, R** 'Implementation of task types in distributed ada' in *Proc. 2nd Int. Workshop on Real-Time Ada Issues* (June 1988)
26 **Gibbons, P B** 'A stub generator for multi-language RP's in heterogeneous environments' *IEEE Trans. Soft. Eng.* Vol 13 No 1 (January 1987) pp 77–87
27 **Magee, J, Kramer, J and Sloman, M** 'Constructing distributed systems in Conic' *IEEE Trans. Soft. Eng.* Vol 15 No 6 (June 1989) pp 663–675