RSD-TR-8-87

AN APPROACH TO DISTRIBUTED EXECUTION OF ADA PROGRAMS

R. A. Volz

P. Krishnan

R. Theriault

Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-2125

April 1987

CENTER FOR RESEARCH ON INTEGRATED MANUFACTURING

Robot Systems Division
College of Engineering
The University of Michigan
Ann Arbor, Michigan 49109-2125

TABLE OF CONTENTS

1. Introduction 1

2. Overview of The Problems And Approach 3

3. Translation Strategy 5

4. Discussion of The Approach 15

5. Status and Conclusion 17

6. Bibliography 18

AN APPROACH TO DISTRIBUTED EXECUTION OF ADA[1] PROGRAMS [2]

R. A. Volz
P. Krishnan
R. Theriault

The Robotics Research Laboratory
Electrical Engineering and Computer Science Dept.
The University of Michigan
Ann Arbor, Michigan 48109

Abstract

Intelligent control of the Space Station will require the coordinated execution of computer programs across a substantial number of computing elements. It will be important to develop large subsets of these programs in the form of a single program which executes in a distributed fashion across a number of processors. In this paper we describe a translation strategy for distributed execution of Ada programs in which library packages and subprograms may be distributed. A preliminary version of the translator is operational. Simple data objects (no records or arrays as yet), subprograms, and static tasks may be referenced remotely.

1. INTRODUCTION

Intelligent control of the Space Station will require the coordinated execution of computer programs across a substantial number of computing elements. It will be important to develop large subsets of these programs in the form of a single program which executes in a distributed fashion acorss a number of processors. The single program approach to programming closely coordinated actions of multiple computers allows the advantages of language level software engineering developments, e.g., abstract data types, separate compilation of specifications and implementations, and extensive compile time error checking to be fully realized across machine boundaries. In this paper we describe one approach to a translation system for distributed execution of Ada programs. We consider loosely coupled homogeneous

[1] Ada is a registered trademark of the U. S. Government (Ada Joint Program Office)

[2] This work has been sponsored by General Dynamics, General Motors, and NASA

systems in which the program/processor binding is specified within the distributed program (static binding).

There have been a number of proposals for distributed programming languages, [1], [2], [3], [4], [5], [6], [7],& [8] to name but a few. Most of these language proposals have emphasized models for communication and synchronization and/or a unified treatment of data abstractions and multi-processing that are amenable to correctness proving. With few exceptions, e.g. [7], however, they have considered neither the real-time aspects of the languages nor the full problem space (see [9]) involved in distributed execution. Only a few of these languages likely to see widespread adoption and use, though of the important principals they lay down are likely be adopted in future language designs.

Ada, on the other hand, will see widespread use, and explicitly admits distributed execution. As yet, there have been only a few attempts at actually distributing its execution. Tedd, et. al. [10], describe an approach that is based upon clustering resources into tightly coupled nodes (shared bus) having digital communications among the nodes. They then limit the language definition for inter-node operations (e.g., no shared variables on cross node references); they are currently in the process of implementing their approach. Cornhill has introduced the notion of a separate partitioning language [11] that can be used to describe how a program is to be partitioned after the program is written. [12] describes this language in greater detail again neither of these approaches recognizes the full problem space involved in the distributed execution of programs. Again, neither of these approaches recognizes the full problem space involved in the distributed execution of programs. [9] describes a number of difficulties which both approaches must face if they are to remain within the current Ada definition.

[9] introduces three major dimensions to the problem of distributed program execution: the memory access architecture, the binding time, and the degree of homogeneity. The range of distributed execution systems that can be represented by these dimensions is larger than any of the distributed language efforts to date, including Ada, can address. For example, one of the major design decisions that must be made in a distributed language is the units of the language that may be distributed. It is likely that one will want to make the distributable units a function of these dimensions. For example, one may want to allow shared variables for some architectures and disallow them for others. [10] does this by disallowing shared variables across node boundaries. Yet the Ada Reference Manual is not clear on this point. There are also questions of to what extent the specification of distribution should be part of the language (as opposed to being stated in a separate configuration phase); [9] discusses these and several other issues. It is clear that the Ada language definition is really not complete with respect distributed execution.

The work described in this paper is, thus, principally an experimental device to help identify the basic issues and point toward their solution. We restrict consideration to homogeneous loosely coupled computers with static binding specified within

the program (one point in the problem space indentified in [9]), and follow the suggestion in [9] and allow only library packages and subprograms to be distributed. We endeavor to avoid placing any further restrictions on the language, and study the implications of effecting distributed execution within these constraints. This implies the need for remote access to data objects, subprograms and tasks.

Rather than write a complete Ada translator, we adopt a pretranslator approach in which we translate a distributed Ada program (- in our system a distributed Ada program is a normal Ada program with SITE pragmas indicating where units are to be placed) into a set of normal Ada programs. We then use existing Ada compilers to translate each of the programs in the set. This approach has the dual goals of being a simpler experimental mechanism and utilizing existing work where possible. It also has a few limitations, which will be pointed out in the remainder of the paper.

The next section presents an overview of the approach. Section 3 then discusses the critical problems and the details of the approach taken. Section 4 analyzes the project performance of the method and section 5 summarizes the current status of the work and discusses directions that must be explored in the future.

2. OVERVIEW OF THE PROBLEM AND APPROACH

The Problem

We presume that the computers upon which a program is to be distributed are interconnected by a communication network, as shown in figure 1. Since we are allowing distribution of library packages and subprograms, our translation system must provide a means of accomplishing the following remote operations:

- Access to procedures and functions declared in remote library units,
- Reading and writing of data objects declared in remote library packages (and hence stored remotely), i.e., remotely shared data is allowed in our model,
- Making [timed/conditional] entry calls on tasks declared in remote library packages,
- Declaring/allocating (local) variables whose types are declared in remote library packages,
- Elaborating tasks whose types are declared in remote library packages,
- Managing task termination for tasks elaborated across machine boundaries.

The approach

The first issue that must be considered is the representation of the distribution. In our system, we write a single program and place a pragma called SITE before each library unit to specify the location on which that library unit is to reside. For example, consider a mobile space robot system consisting of several mobile vehicles (each with a robot mounted on it) and an overall system controller. If it were desired to have one vehicle controlled by computer number 2 and the overall control using it (as well as several other similar systems) placed on computer 1, a sample of the relevant code would look as follows:

```
pragma SITE (2);
package VEHICLE is
    procedure MOVE(..);
          :
end VEHICLE;
```

```
pragma SITE(1);
with VEHICLE;
procedure CONTROL is
   :
begin
   :
   VEHICLE.MOVE(..);
   :
end CONTROL;
```

Figure 2 illustrates the overall operation of our system. A pre-translator translates our single Ada program with SITE pragmas into a set of independent Ada programs which include library modules of our design to effect communication among processes. The translation system would replace all calls (within CONTROL) to the procedure VEHICLE.MOVE(..) with the appropriate remote procedure call. Similarly any references in CONTROL to data objects or task entries defined in package VEHICLE would be translated into appropriate remote references. Each of the individual programs can thus be compiled by an existing Ada compiler. This approach simplifies the translation process considerably since our pre-translator is much less complex than a full Ada compiler.

The remote operations and interprocess communication are managed by a set of *agents* created for units that can be remotely referenced and an underlying process to process mailbox system: During the pass through the pre-translator, all references to remote objects are replaced with appropriate references to *agents*. Each program unit (for example, suppose it is unit A) which might be referenced from another,

remote, unit (call it B) has three categories of agents associated with it. There is a *local agent* on the site holding A, a *remote agent* on the site holding B (and every other site referencing A), and a *pointer agent* placed on the same site as the local agent. A is essentially unchanged by the pre-translator, and both the local and remote agents can be generated from only the specification of A. All references within B to code or data objects in A are translated into appropriate calls to the remote agent of A (which resides on the same site as B), which in turn uses the message system to pass the service request to the local agent of A. The local agent performs the necessary functions, returning any objects requested. The pointer agent serves to propagate object accesses involving access variables pointing to other sites, and is only required if such access variables are used.

The relevant entities and flow of data for the example above are shown in Figure 3. CONTROL_T is the translation of CONTROL with the references to objects in VEHICLE replaced with calls into VEHICLE_REM_AGENT.

3. TRANSLATION STRATEGY

In order to solve the problems raised in the previous section the following issues must be resolved:

- development of a general remote object accessing methodology,
- translation of source code references to remote objects,
- management of other remote service functions, e.g., creating objects, and
- generation of the agents.

The solution to these problems, while leading to reasonably efficient code, involves a rather complex set of multiple pass operations and the generation and use of a number of auxiliary files of intermediate information. Thus, a set of utilities also are needed to allow the user to perform these operations in a straightforward manner.

By far the most complex of these issues is the development of a general remote object accessing method. This is complicated by the need to address arbitrarily nested record and array components and the fact that component pointers may point to logically nested records or arrays on other processors. We thus concentrate our discussion on matters relating to object access. The solution to most of the other issues follows the resolution of this problem in a reasonably straightforward manner.

The structure of the agents is critical to solving this problem, and is generally in three parts: 1) elements to access code objects, 2) elements to manage the address chain leading through qualified names of records and arrays, and 3) elements to manage other services. The interprocessor mail system and message structure is closely integrated with the agent structure.

We begin this section with a discussion of the overall agent structure and its use for accessing code objects, and then discuss related important issues of access via fully qualified names, the postal message structure, and the translation process.

Agent Structure

As mentioned in the previous section, three kinds of agents are generated whenever a library unit specification is encountered by the pretranslator: a local agent, a remote agent, and a pointer agent. Each agent is generated as a separate package, and assigned a unique name that is derived from the source package name. The agents can be generated simply from the package or subprogram specifications.

The terms *local* and *remote* agent are used with respect to the processor holding the library unit which they represent. That is, the local agent resides on the same node as the unit it represents, while the remote agent resides at each other node referencing the unit. Thus, a remote action of some kind begins with the referencing unit making a call (after it is processed by the pre-translator) to the remote agent of the unit being accessed. For example, if the cell controller CONTROL makes a call to VEHICLE.MOVE(..), the translated procedure CONTROL_T makes a call to the remote agent VEHICLE_REM_AGENT. We thus consider remote agents first.

Remote Agents

Remote agents are merely collections of procedures and functions that effect remote calls. In the case of subprogram and task entry calls, they present an interface to the calling package identical to that of the original source package. In our previous example, the package VEHICLE_REM_AGENT will contain a procedure MOVE(..) that will receive the call intended for package VEHICLE. These procedures and functions format an appropriate message record (described below), and dispatch it to the appropriate site via the postal service. When a return value is received from the local agent (on the other processor) via the postal service, this value will be 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. Instead, a set of procedures to get and put the values of remote objects of various types is generated. Again, these procedures are generated from the specification of the library package being referenced. In this case, the referencing unit (CONTROL in our previous example) must be translated to replace the object reference with a call to the appropriate get or put routine. Since the get and put routines can be overloaded (with respect to the specific argument types used) the

translation is straightforward. The specific arguments used and the detailed actions of the get and put routines are closely intertwined with the management of fully qualified names, and will be discussed later.

Two other points are worth mentioning. Since the structure of the remote agent was chosen to minimize the impact on the referencing unit, the translation required by the pretranslator is a minimum. Also, since sending and receiving messages from another processor is time consuming (relative to normal instruction processing times), the point after the transmission of a message is treated as a sychronization point so that other tasks may obtain the services of the cpu while the reply to the message is in progress.

Local Agents

The local agents are the most complicated of the three agent types. Their task is to service requests from remote sites needing to access data objects, subprograms, or task entries. A local agent consists of N+2 tasks where N is the total number of functions, procedures, and task entries, contained in the source specification of the unit the agent is helping to represent. One of these tasks is associated with each of the aforementioned subprograms and task entries.

One of the remaining two tasks is designated as the local agent *main* task. This task consists of a single loop that requests message records from the postal service (via a task entry call), interprets the request, and dispatches the request to the appropriate handler (task or procedure) within the local agent. Messages requesting access to data objects, are serviced immediately within the main task by calling a GETPUT procedure (described below) and an immediate reply is sent.

```
with PACKAGE_NAME;                    package being represented
task body AGENT_MAIN is
   M: MESSAGE_TYPE;
begin
   loop
      POSTAL.MAIL_BOX.GET(M); - - get message
      case M.OBJ_ENUM is              - - Branch according to object name.
         - - Object references
         when NAME_1 => GETPUT(M, PACKAGE_NAME.NAME_1);
                  ⋮
         when NAME_K => GETPUT(M, PACKAGE_NAME.NAME_K);
         - - Subprogram calls and task entries
         when NAME_K1 => MANAGER.DEPOSIT_NAME_K1(M);
                  ⋮
         when NAME_N => MANAGER.DEPOSIT_NAME_N(M);
```

```
        end case;
        SEND_RETURN(M);
    end loop;
end AGENT_MAIN;
```

The above abstraction is only for a single distributed package. Actually, the message type would be embedded in a yet more general record having a varient part for each distributed package, and the actual code would be slightly more involved.

It is imperative that the main task not be blocked for it provides concurrent access to all objects and types in the specification of the unit it represents, and if blocking occurred here, other, parallel, requests could be delayed. In particular, the agent must not be blocked by a unit it calls on behalf of a remote client, as could occur if the agent directly called the unit (the subprogram called might, for instance, become blocked on an I/O wait). That is why a *task* is associated with each subprogram and task entry. The main task places the message received in a buffer, by calling an entry in a buffer manager task (the last of the tasks in the local agent). A flag counter corresponding to the requested call is also incremented at this time.

The buffer manager task has N additional entries, whose acceptances are conditioned on a positive value of each of the corresponding N counters (indicating that there is a message to be retrieved).. There are N call manager tasks (the N tasks corresponding to subprograms and task entries), whose sole purpose is to retrieve a message record from the corresponding conditional task entry in the buffer manager task, execute the appropriate call to the source package body, and then return a reply to the remote site.

The following code abstraction illustrates the manager task and the tasks corresponding to the subprograms and task entries that may be called.

```
task MANAGER is
    entry DEPOSIT_E1(MESG : in MESSAGE);
    entry DEPOSIT_S1(MESG : in MESSAGE);
            ⋮
    entry EXTRACT_E1(MESG : out MESSAGE);
    entry EXTRACT_S1(MESG : out MESSAGE);
            ⋮
end;

task body MANAGER is
    E_FLAG: array(1..MAX_ENTRIES) of INTEGER;
begin
    loop
```

```
        select
            accept DEPOSIT_E1(MESG : in MESSAGE) do
                - - deposit the message for e1
                E_FLAG(1) := E_FLAG(1) + 1;
            end;
                        ⋮
        or
            when E_FLAG(1) > 0 =>
                accept EXTRACT_E1(MESG : out MESSAGE) do
                    - - extract a message from the buffer and return it
                    E_FLAG(1) := E_FLAG(1) - 1
                end;
                        ⋮
        end select;
    end loop;
end MANAGER;
```

The suffix EI indicates the Ith entry point, and the suffix SI indicates the Ith subprogram. The structure of the entry task for entry E1 is as follows:

```
task DO_E1
begin
    loop
        MANAGER.EXTRACT_E1( );
            E1( );
            - - send back message;
    end loop;
end DO_E1;
```

The messages are provided by a mailbox system that delivers messages to the correct local agent. The message interpretation and task calls by the agent essentially achieves a routine to routine communication between routines in the remote and local agents in a way that prevents delays in the response to one request from locking out other parallel requests.

Pointer agents

Our allowed model of distribution permits access variables to point to 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. 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.

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 site which it does not directly reference and for which it does not therefore have an agent. We therefore include pointer agents to allow access to objects on remote sites. The structure of pointer agents is similar to that of local agents, except that provision for subprogram calls need not be made.

Remote Data Object Access

Three characteristics of Ada data objects cause difficulty in developing a general mechanism for handling references to remote objects: 1) the objects may be composite objects, 2) they may have concatenated names, and 3) parts of a fully concatenated name may be access variables pointing to objects on other machines.

The first issue manifests itself when one must copy a composite object (as opposed to a component of the object) from one site to another. For example, suppose that site 2 uses a record A on the right hand of an assignment statement and that A is located on site 1. Eventually, the agent and message system must convert A to a bit string for transmission. It would usually be desireable that the part of the system that performs the conversion not be aware of the structure of the object (from object oriented design principles). However, if the object contains a memory address as part of its structure, the result received could be meaningless. For example, suppose the record A contains a variable length array, as shown below.

```
subtype S is INTEGER range 1..MAX;
type IA is array (INTEGER range <>) of INTEGER;
type R(L: S := 1) is
     record
          B: IA(1..L);
          C: INTEGER := 0;
     end record;
A: R;
```

One decision for the memory allocation for the record might be to allocate the

storage for the array from a heap and place only a pointer to the array (or possibly its dope vector) in the record. The need to perform whole object (record) assignments in Ada might discourage such a memory allocation scheme, but nevertheless, it is certainly a possibility. A bit by bit copy of the block of data corresponding to the record A, would then copy this address, which would have no usefulness when received by the requesting unit; in particular, the bit by bit copy of the record block would not result in the array values being transmitted. To avoid this problem, the routine that does the final message transmission must, indeed, contrary to the above assumption, have knowledge of the record structure so that the array values themselves may be transmitted, and not just the address of the array. Since we are describing a pre-translator approach that uses existing Ada compilers, this knowledge is dependent upon the implementation of the underlying compilers used.

To see the second issue, suppose that site 2 contains a statement like $X := A.C$. How does one construct an address for A.C? Or describe, in a general way, to the agents what element is to be returned? The syntax "A.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 A, not its physical structure. Again, implementation dependent knowledge of the rules used for construction of the physical structure of records is necessary.

If one were to now add a fourth component, D, to the type R above, that is an access type, and if the value of A.D were to point to another record stored on site 3, the third issue arises. The method used to calculate the address of the item to be retrieved must not only contain implementation dependent knowledge, but it must be distributed as well.

Strategies for Remote Object Access

We are studying two methods of obtaining composite (as well as scalar) objects: 1) using knowledge of the rules for storage allocation and physcial record and array construction, develop the distributed algorithms for calculating the address of the target object and then implement these, possibly in assembly or some other lower level language, and 2) use minimal implementation dependent knowledge and the logical structure of records and arrays to utilize standard Ada mechanisms to perform the object transfers. We expect the former to lead to more compact (in terms of code size) solutions, but to require a more detailed knowledge of the internal workings of the underlying compilers, while the latter will require less knowledge of the internal mechanisms used by the compilers at the expense of a larger amount of code (automatically generated, however) in the agents. Since the latter is also more in keeping with the philosophy of using existing compilers where possible with minimal knowledge of their internals, and since developing this approach will aid in developing the algorithms for the first approach, we have followed this one first, and it is this one that will be described below. In subsequent work, we will explore

the direct calculation of object addresses.

Access to remote objects is based upon the following things:

- An enumerated type, T_ENUM, whose values are the names of every type and field declared in the package for which an agent is being generated, and those in packages included via a with.

- An enumerated type, N_ENUM, whose values are the names of every data object declared in the package for which an agent is being generated, and those in packages included via a with.

- A collection of GETPUT procedures, one for each record or array type defined, whose functions are to either handle the request for an object reference if the request is for an object of the type the GETPUT handles, or to call another GETPUT if the object requested is, or is derived from, one of the fields of the type.

- A variant message structure containing appropriate fields indicating the type of data required, the fields within records to be used, and an actual data object of the type being referenced.

From the perspective of the local agent, a remote direct (not via access variables) data object access begins with the local agent main task receving a message from the postal system. One of the fields in this record contains a value of type N_ENUM that indicates the outermost name in the fully qualified name of the object being referenced. The local agent main task then performs a case statement on this value. There is thus a case for each object name. Each case calls a GETPUT procedure and passes it the message, the object named, and a count of the number of name components to the fully contatenated 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 passed to it. Another field in the message record contains the type of the object to be returned.

If the COUNT is not zero, then either an array element is being sought, or a fully concatenated name has not yet been fully expanded. In the former case, the indices for the array element (or slice) are contained in other fields of the message record and the GETPUT can select the appropriate element(s) of the array. These either directly satisfy the request or are used to recurse as described next.

If the GETPUT is handling a record type, there will be another field in the message record corresponding to this type of record which will contain a value of type T_ENUM (containing the field name to be selected). The GETPUT contains a case statement conditioned on this field indicator. There is thus a case corresponding to each field possible in the record. The action of each branch of the case is similar.

Another GETPUT is called, passing to it the message record and object pointed to by a concatenation of the object name passed in and the corresponding field name.

Below is an abstraction of a typical GETPUT routine for a record type. The forms for other types are similar, but tend to be even a bit simpler.

```
procedure GETPUT(M: in out MESSAGE; OBJ: in out T; COUNT: NATURAL) is
begin
   COUNT := COUNT - 1;
   if COUNT = 0 then              - - the name is fully expanded
      if <a get request> then
         - - copy value from OBJ to appropriate field in message record;
      else
         - - copy value from appropriate field in message record to OBJ;
      end if;
      return;
   end if;
   case <field name from message record> is
      when F1 => GETPUT(M, OBJ.F1,COUNT);
                 :
      when FN => GETPUT(M, OBJ.FN,COUNT);
   end case;
end;
```

Here T is a record type of an object being passed in and F1..FN are the fields in the record type. If one of the fields, FI, say, were an access variable, that access variable would have been replaced by a record (as described in the pointer agent section above) and the action for the corresponding case would first check to see if the requested object were 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 pointer agent on the indicated site.

Message Record Structure

The interprocessor message structure is key to the operation of the above object referencing scheme. For each source package, a different message record type is defined. These records consist of a fixed part, and a variant part. There is one case of the variant part for each type of data object defined in the source package. In the case of a subprogram or task entry call, the variant part of the record contains fields for all of the arguments, and if applicable, a function result. The fixed part of the record contains field selectors which are used for accessing fields of records, as described above. A simple example of a message record type is given below. It should be self-explainatory from the previous discussion.

```
type MESS_T( DATA_TYPE: T_ENUM ) is
   record
      OBJ_ENUM : N_ENUM ;          - - indicates outermost object
      TYPE1_FIELD : T_ENUM;        - - record type TYPE1
      TYPE2_X1 : TYPE2_X1_T;       - - 2-dim array type TYPE2
      TYPE2_X2 : TYPE2_X2_T;
            :
      case DATA_TYPE is            - - reflects data to be exchanged
         when TYPE1_D =>
            TYPE1_VAL : TYPE1;
         when TYPE2_D =>
            TYPE2_VAL : TYPE2;
         when CALL1_D =>              - - function CALL1
            CALL1_ARG1 : FLOAT;
            CALL1_RESULT : FLOAT;
         when CALL2_D =>              - - subprogram CALL2
            CALL2_ARG1 : INTEGER;
            CALL2_ARG2 : INTEGER;
         when FLOAT_D =>
            FLOAT_VAL : FLOAT;
         when INTEGER_D =>
            INTEGER_VAL : INTEGER;
      end case;
   end record;
end;
```

Since the postal service deals with all types of messages, a global message record type is defined. The global message record also consists of a fixed part, and a variant part. The various cases of the variant part are, as one might guess, merely the different message records for each source package. The fixed part contains the destination package number, and the return address, which consists of the source site number, and a logical channel number.

Translation Procedure

The translations required for the methods outlined above involve numerous steps and are quite involved. In this section we describe briefly the procedures to be used and a utility that has been prepared to simplify use of the pre-translator.

The first step in the translation procedure is to insure that the program to be distributed is correct. This is accomplished by compiling it for a single system. The programmer must do this before invoking the pre-translator.

When a correct program is available, 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) pre-link operations, 4) determination of the order of compilation of original sources (including agents) for target sites, 5) compiling and linking of individual site programs. Two utilities have been written to facilitate some of these steps.

The pre-compilation utility (ADAUTIL) will translate 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 must be specified, and one or more targets (main programs) must be specified. Since the order of pretranslation is identical to the order of Ada compilation, ADAUTIL takes an option specifying whether a makefile to run the pretranslator, or a makefile to run the Ada compiler is desired.

The second utility, called MESSUTIL, performs step three above. The operations done during setp 3 are: 1) constructing the global message record from all relevant package message records, 2) constructing a package of package site constants, 3) constructing main procedures for each site, and 4) constructing a meta makefile capable of performing steps 4 and 5 above.

Two scripts were written to simplify the pretranslation process. One script performs steps 1 to 3 above, and the other invokes the meta makefile, to perform steps 4 and 5. If any non-Ada object modules need to be linked into any site, the meta makefile may be edited in between the running of the two scripts.

4. DISCUSSION OF THE APPROACH

One of our principal concerns with the system developed is the run-time overhead associated with the mechanisms we used. We can model this performance in terms of the run-time overhead associated with various kinds of remote references. From the tests performed in [13] we know that task rendezvous times exceed procedure call times by one and a half to two orders of magnitude, and that task elaboration times are several times larger than rendezvous times. We can also reasonably expect the network communications times to be sizable. For example message end-to-end times for MAP are on the order of 100ms, more or less independent of message size [14], for the Intel hypercube, a few milliseconds, and for the NCUBE hypercube, several hundred microseconds to a millisecond, where the latter two depend somewhat upon message size, the variable component of message size being 1-10 microseconds/byte [15]. We thus neglect all local procedure and function call times, and model our overhead in terms of the number of messages and rendezvous required.

Thus, let t_m and t_r be the times to complete a message transfer and local rendezvous, respectively and let n_m^o and n_r^o be the number of messages and local rendezvous required for a remote operation of type o. Then, the time to complete a

remote operation is

$$n_m^o \cdot T_m + n_r^o \cdot t_r$$

In these cases, we represent the overhead by the pair (n_m^o, n_r^o).

Whenever there are task elaborations involved, we represent the number by E. It is listed separately since it is generally not necessary to do the task elaboration with each access, but only when tasks or procedures are first elaborated. Nevertheless, even though many of these need be done only once immediately after system load, the number of tasks in the system could have an impact on the scheduling algorithms to be used and the efficiency of any runtime system, and the number E is thus important.

The following sections present briefly the costs associated with each of the remote operations.

Data Objects — (2,4), E = 0

Access/Updates to data objects require two messages and four rendezvous. One message is to send the request and the second to receive an acknowledgement. The rendezvous are for the mail system. This presumes that the requested object is on the first remote site accessed. If there is a continuation to other sites through pointers, the above numbers must be mulitplied by the number of remote accesses required to satisfy the request.

Task Objects — (2, 6), E = # of entries

Task objects are accessed through entry calls. This requires two messages as for data objects and six rendezvous for synchronization (4 for the mail system and 2 for the handler).

The number of task elaborations that need to be done initially is equal to the number of entries to the task. Entry calls to task objects created from task types require no special handling by themselves. However, each task object created from a remote type requires two messages for creation and four rendezvous for synchronization. All further access are as in the case of task objects.

Procedures and Functions —(2, 6); E = 1

Since the local agent treats procedure and function calls in the same way as task entry calls, the analysis is analogous.

Pointers

There are two factors to consider here, the overhead when the object pointed to is remote, and the overhead when the object is local. Remember that all pointers

are replaced with records having a site number and a pointer. This requires that all accesses via pointers begin with a check of whether or not the object is local or remote. If remote, the time of the check will be insignificant in comparison to the time required for the remote access and may be neglected. In this case the overhead depends upon the type of objected being referenced, and will follow the results obtained above.

However, if the access is local, the overhead is more significant. The exact amount of degradation will depend upon how an individual compiler implements pointer accesses and if then else constructs. In a simple test in which we wrote as efficient assembly language code as we could for local pointer accesses with and without the pointer record construct used here, the difference was a factor of four. In interpreting this, however, one must take into account the magnitude of time involved (only a few microseconds are the most) and the frequency of occurence. With these considerations taken into account, we do not feel that much overall time will be added to local accesses.

Summary Analysis Comments

To place the above analyses in perspective, one must compare typical times for message transfers and rendezvous. Some typical network times were mentioned above. Rendezvous times on the order of 500-600 microseconds have been reported for an 8 mhz IBM PC/AT, and on the order of 300-400 microseconds for Motorola 68000 processors. It is also the case that these times have been dropping significantly with each new release of Ada compilers intended for real-time applications, and are predicted by Ada vendors to become yet considerably smaller over the next year or two. Thus, except for the fastest networks, the message times will either be close to the rendezvous times or dominate them, and the approach taken will be primarily influenced by the network message times.

There is further issue that may be of concern, the number of tasks and GETPUT routines needed in the local agents. These have a linear dependence upon the number of entries (and subprograms) and types present in a remotely accessed package. While this may seem rather large, one is not likely to access a large number of things remotely, and those that are accessed remotely can be packaged separately from those that are not, thus keeping the number of extra tasks and routines to a minimum.

5. STATUS AND CONCLUSIONS

At the present time, the distributed translation system is operational for distributed packages with simple objects in their visible parts, i.e., no record or array definitions. Scalar data objects, subprograms and declared tasks may be directly referenced (no timed or contional calls). Tests have been successfully completed

with up to three VAX processors cooperating on the execution of a single program. The implementation of the strategy described for referencing arrays and records (with fully concatenated names) is nearly complete, and expected to be in operation within a few weeks.

Nevertheless, there is still considerable work to be accomplished before the distribution of library packages and subprograms is complete. Although the strategy has been determined (see [16]), work has not yet been begun on handling timed/conditional task entry calls. Similarly, the dynamic creation of tasks is not complete. Two strategies will be implemented in this case. In the first, the created objects will be placed on the site elaborating the definition of the task type. In the second, the task object will be placed on the site creating the task through a declaration or new operator. The first is simpler to implement, but may make the task objects remote from the unit executing the code calling for their creation, while the second implementation is considerably more complex, and as noted in [9], may contain hidden remote object references. Finally, task termination must be properly handled.

More importantly, there are many issues of language definition that must be addressed. Our work has only addressed one point in the problem space to date, homogeneous, loosely coupled systems with static distribution. Additional representation mechanisms are needed to describe limitations dependent upon architectural considerations, to describe binding mechanisms, and to describe processor types (so that implicit data conversions can be accomplished). Moreover, it is probably necessary to require greater use of representational specifications on data objects to which remote access is allowed. Finally, there should be a more explicit definition of the allowed units of distribution.

Bibliography

[1] Hoare, C.A.R., "Communicating sequential processes", *Communications of the ACM*, vol. 21, no. 9, Aug. 1978.

[2] Strom, R.E. and Yemini, S., "NIL: an integrated language and system for distributed programming", *Sigplan '83 Symposium on Programming Language Issues in Software Systems*, vol. 18, no. 6, June, 1983, pp. 73-82.

[3] Liu, M.T. and Chung-Ming Li, "Communication distributed processes: a language concept for distributed programming in local area networks", *Local Networks for Computer Communications, IFIP Working Group 6.4, International Workshop on Local Networks*, Aug., 1980, pp. 375-406.

[4] Van DenBos, J. and Plasmeijer, R. and Stroet, J., "Process comm. based on input specifications", *ACM Trans. of Programming Languages & Systems*, vol. 3, pp. 224-250, July, 1981.

[5] Andrews, G.R., "Synchronizing resources", *ACM Trans. of Programming Languages & Systems*, vol. 3, no. 4, pp. 405-430, Oct., 1981.

[6] Mao, T.W. and Yeh, R.T., "Communication Port: A Language Concept for Current Programming", *IEEE Trans. Software Eng.*, vol. SE-6, no. 2, pp. 194-204, March, 1980.

[7] Holt, R.C., "A short introduction to concurrent euclid", *Sigplan Not.*, vol. 17, no. 5, pp. 60-79, May, 1982.

[8] Hansen, P.B., "Edison-A multiprocessor Language", *Software-Prac. and Exper.*, vol. 11, no. 4, pp. 325-361, April, 1981.

[9] Volz, R.A. and Mudge, T.N. and Buzzard, G.D. and Krishnan, P., "Translation and Execution of Distibuted Ada Programs: Is It Still Ada? ", *IEEE Transactions on Software*, Spring 1987.

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

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

[12] Honeywell Systems Research Center, "The Ada Program Partitioning Language", the Distributed Ada Project, Sept., 1985.

[13] R.M. Clapp, L.J. Duchesneau, R.A. Volz, T.N. Mudge, and T. Schultze, "Toward real-time performance benchmarks for Ada," *Communications ACM*, vol. 29, no. 8, pp. 760–778, Aug. 1986.

[14] Volz, R.A. and Naylor, A.W., *Final Report of the NSF Workshop on Manufacturing Systems Integration*, held November 1985 in St. Clair, Michigan and organized by the Robotic Systems Division, Center for Research on Integrated Manufacturing, College of Engineering, The University of Michigan, Ann Arbor, MI. 48109, 1985

[15] T. N. Mudge, G. D. Buzzard, T. S. Abdel-Rahman, "A High Performance Operating System for the NCUBE," *Proceedings of the 1986 Conference on Hypercube Multiprocessors*, Knoxville, Tennessee, Oct. 1986.

[16] R.A. Volz, and T.N. Mudge, "Timing Issues in the Distributed Execution of Ada Programs," to appear in special issue on Parallel and Distributed Processing, *IEEE Transactions on Computers*, 1987.

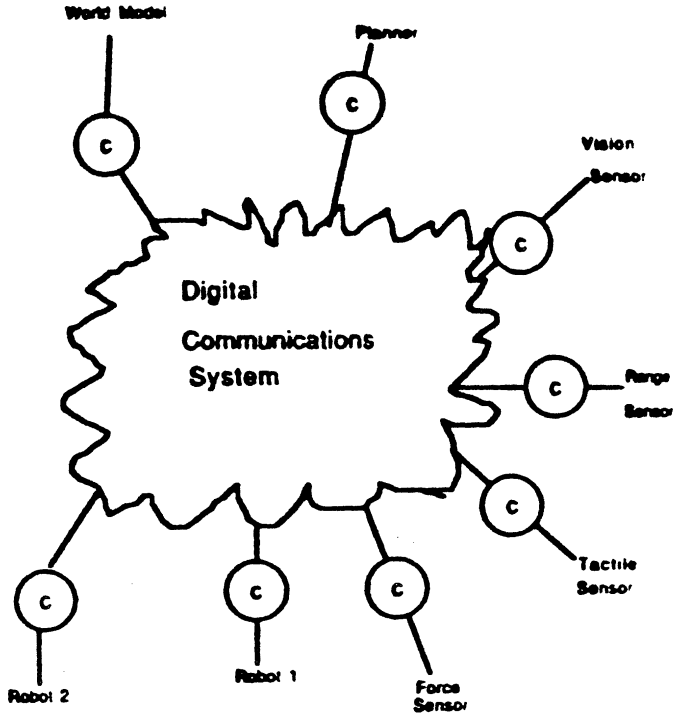

Figure 1: Loosely coupled system upon which we seek distributed program execution

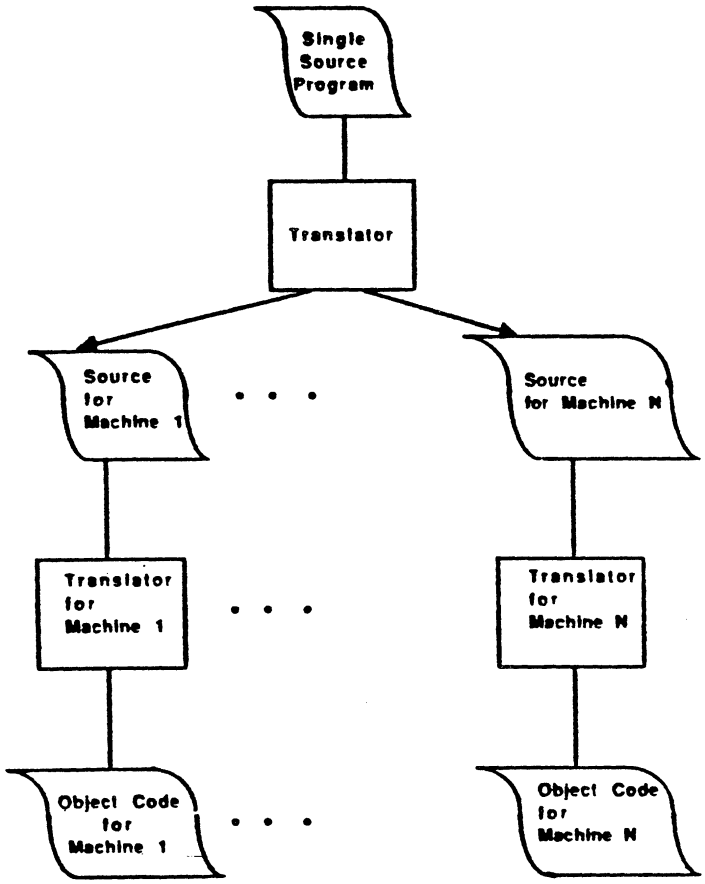

Figure 2: Overall operation of translation system

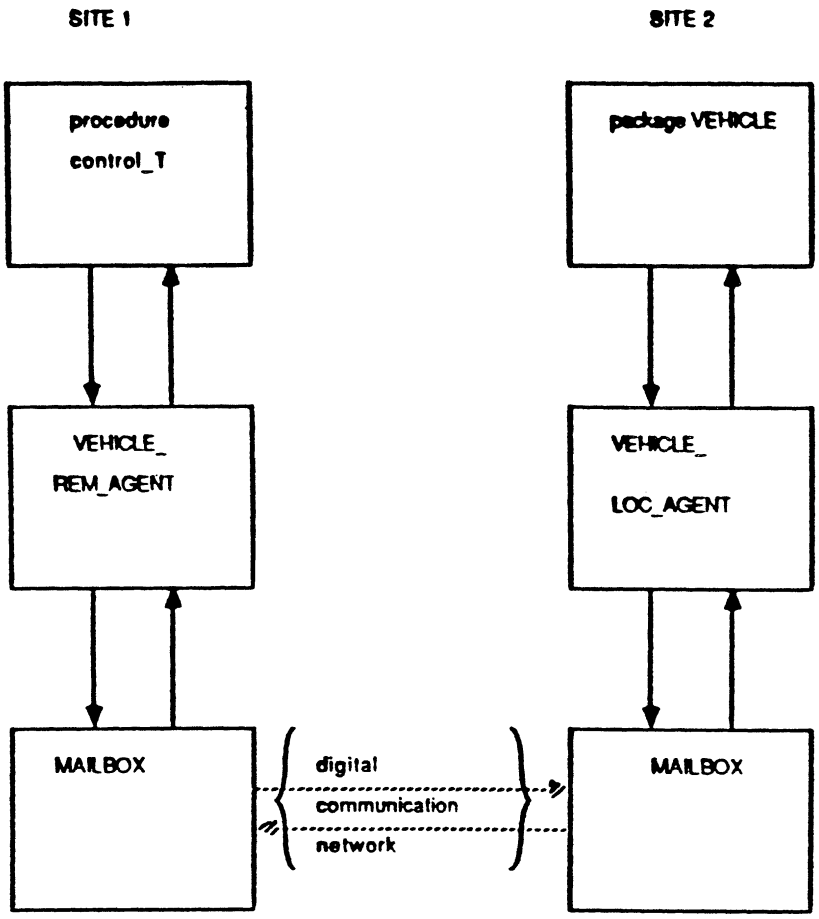

Figure 3: Structure of translated example program