JOHNSON [illegible]
IN-62-CR
96752

Distributed and Parallel Ada and the Ada 9X Recommendations

(NASA-CR-190392) DISTRIBUTED AND PARALLEL Ada AND THE Ada 9X RECOMMENDATIONS (Research Inst. for Computing and Information Systems) 28 p

P.28

N92-26199

Unclas
G3/62 0096752

Richard A. Volz
Stephen J. Goldsack
R. Theriault
Raymond S. Waldrop
A.A. Holzbacher-Valero

Texas A&M University
Imperial College of London

April 1992

Cooperative Agreement NCC 9-16
Research Activity No. SE.35
Deliverable Task R5B Report

NASA Johnson Space Center
Engineering Directorate
Flight Data Systems Division

ricis

Research Institute for Computing and Information Systems
University of Houston-Clear Lake

TECHNICAL REPORT

The RICIS Concept

The University of Houston-Clear Lake established the Research Institute for Computing and Information Systems (RICIS) in 1986 to encourage the NASA Johnson Space Center (JSC) and local industry to actively support research in the computing and information sciences. As part of this endeavor, UHCL proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a continuing cooperative agreement with UHCL beginning in May 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The UHCL/RICIS mission is to conduct, coordinate, and disseminate research and professional level education in computing and information systems to serve the needs of the government, industry, community and academia. RICIS combines resources of UHCL and its gateway affiliates to research and develop materials, prototypes and publications on topics of mutual interest to its sponsors and researchers. Within UHCL, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business and Public Administration, Education, Human Sciences and Humanities, and Natural and Applied Sciences. RICIS also collaborates with industry in a companion program. This program is focused on serving the research and advanced development needs of industry.

Moreover, UHCL established relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research. For example, UHCL has entered into a special partnership with Texas A&M University to help oversee RICIS research and education programs, while other research organizations are involved via the "gateway" concept.

A major role of RICIS then is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. RICIS, working jointly with its sponsors, advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research and integrates technical results into the goals of UHCL, NASA/JSC and industry.

Distributed and Parallel Ada and the Ada 9X Recommendations

RICIS Preface

This research was conducted under auspices of the Research Institute for Computing and Information Systems by Richard A. Volz, R. Theriault and Raymond S. Waldrop of Texas A&M University and Stephen J. Goldsack and A. A. Holzbacher-Valero of Imperial College, London, England. Dr. E.T. Dickerson served as RICIS research coordinator.

Funding was provided by the Engineering Directorate, NASA/JSC through Cooperative Agreement NCC 9-16 between the NASA Johnson Space Center and the University of Houston-Clear Lake. The NASA technical monitor for this activity was Terry D. Humphrey of the Systems Software Section, Flight Data Systems Division, Engineering Directorate, NASA/JSC.

Distributed and Parallel Ada and the Ada 9X Recommendations

Task R5B Report

NASA Subcontract #074
Cooperative Agreement NCC-9-16
Research Activity # SE.35

Period of Performance: May 1, 1990 - September 30, 1992

Submitted to
RICIS

Submitted by
R. Volz, Texas A&M University
S. J. Goldsack, Imperial College, London, England
R. Theriault, Texas A&M University
R. Waldrop, Texas A&M University
A. A. Holzbacher-Valero, Imperial College, London, England

Distributed and Parallel Ada and the Ada 9X Recommendations[1]

by

Richard A. Volz[2]
Stephen J. Goldsack[3]
Ronald Theriault[4]
Raymond Waldrop[5]
Alexandra A. Holzbacher-Valero[6]

1 Introduction

The Ada programming language was developed around 1980 to become the standard language for use in the field of embedded computer systems, particularly in work for the United States Department of Defense. The ISO standard version of the language was established in 1983, and is commonly referred to as Ada83. This language provided no special features to support the construction of distributed systems, and much research has been devoted to finding suitable ways of writing such programs in the language. Recently, the DoD has sponsored work towards a new version of Ada intended to overcome some of the recognized shortcomings of Ada83. The revised version, often called Ada9x[1], will become the new standard sometime in the 1990s. It is intended that Ada9x should provide language features giving limited support for distributed system construction. The present paper will review the requirements for such features, especially in the light of the the Ada9x mapping proposals published in August 1991.

Many of the most advanced computer applications involve embedded systems that are comprised of parallel processors or networks of distributed computers. If Ada is to become the widely adopted language envisioned by many, it is essential that suitable compilers and tools be available to facilitate the creation of distributed and parallel Ada programs for these applications. Given the "no supersets, no subsets" philosophy of Ada, the creation of these tools poses some challenging problems.

It has been argued [2] that matters of distribution depend upon the architecture of the system. [3] carries this a step further and argues that any parallel or distributed system is comprised of a set of memories and a set of processors, and that it is appropriate to partition the system by the subsets of processors sharing each memory.

[1]This work was supported by NASA under Cooperative Agreement number NCC-9-16 and by Telesoft under Contract number 91-573

[2]The Computer Science Department, Texas A&M University

[3]Imperial College of London

[4]The Computer Science Department, Texas A&M University

[5]The Computer Science Department, Texas A&M University

[6]Imperial College of London

Accordingly, when we use the term"processor" in this document, we actually mean the set (possibly singleton) of processors sharing some common memory. Further, [3] argues that the present Ada tasks are the natural unit of distribution among processors sharing the same memory, but that extended considerations are necessary for non-shared memory. We thus focus this study on non-shared memory distributed and parallel computer systems, as extensions to shared memory (sub)systems is relatively straightforward.

In particular, we discuss the major issues that must be resolved and describe alternative solutions. It is our intent to concentrate on the high level issues and decisions that must be made, not to trace out any one possible solution in complete detail. However, enough detail is provided so that it should be possible for an informed reader to synthesize the various specific solutions.

We are specifically interested in reconciling alternative techniques for distributed and parallel programming with the draft proposals for Ada 9X. It is our suggestion that an extended typing model provides a consistent framework within which to develop Ada to support distributed and parallel systems. Further, through replication (controlled through a configuration tool external to the language itself) of appropriately restricted units presently in Ada, e.g., Pure packages of Ada 9X [1], reasonable distributed and parallel extensions can be created without introducing new typing syntax into the language. It is only necessary that replication of suitable units be allowed. We find that with a little tuning (principally permitting replication of Remote Call Interface packages and/or remote calls to pure packages), the Active Partitions proposed in the August, 1991 draft Mapping Specification can provide a base upon which reasonable solutions can be developed. The proposal for parallel tasks, however, is less clear.

While we explicitly consider only homogeneous systems here, many of the techniques used apply equally well to heterogeneous systems. For a more complete treatment of heterogeneous systems, see [4].

In the following sections, we introduce the major issues, discuss alternative strategies for developing distributed and parallel Ada, and make a set of recommendations for minor modifications to Ada 9X we believe would better support distributed and parallel computation.

2 Issues and Discussion

2.1 Basic Capabilities Required

First, we examine the similarities and differences between distributed and non-shared memory parallel processing systems and argue that the same language capabilities are needed for both.

One usually interprets the term "distributed system" to mean a modest (typically two or three up to a few dozen) set of loosely coupled processors that may be either homogeneous or heterogeneous. Non-shared memory parallel computers, on the other hand, typically have 100's to 1000's of homogeneous processors. Moreover, this number may change in different runs of the program, possibly dynamically. One often (not necessarily correctly) assumes that the processors in a distributed system are likely to be doing fundamentally different things, whereas many, if not all, of the processors in a parallel computer are doing very similar things. Perhaps as a result of these differences, distribution has sometimes been thought of in terms of distributing different elements of a program without replication of any units, while parallelism has been thought of in terms of replicating some elements of a program. We shall argue that both distributed and parallel systems need both capabilities.

For parallel machines, it unreasonable for the programmer to have to explicitly create 100's or 1000's of units to be placed on the different processors. This implies that some form of typing construct, or replication, must be available from which [indexed] instances of some unit of code can be created on different processors. These instances must be callable from remote processors. However, one of the principal advantages of Multiple-Instruction-Multiple-Data (MIMD) machines is that one does not need to perform the same functions on every processor. Some method is needed to distribute these functions among the processors of the machine.

For distributed systems, on the other hand, some have put forth distribution mechanisms without a typing or replication capability. We believe, nevertheless, that some form of typing construct or replication of the units of distribution is needed. Indeed, we believe that two different forms of replication are needed. Reasons for replication appear frequently in the real world - objects modelled or controlled are replicated, e.g., bank tills in a banking system, check-out stations in a supermarket, multiple elevators in a building. Thus, embedded systems need replication.

The first form of replication needed is for local (to a given processor) use of a common utility, and does not involve remote calls to the replicated units. Servers such as TEXT_IO, CLOCK and math packages are examples of entities that may well be needed on different processors in the system. Each processor would have its own version.

The second form of replication is of the remotely callable functions being distributed. This may be necessary either because several processors are performing the same function with different external devices or data, or for fault tolerance reasons. For example, one distributed system under study has 20 identical operator consoles, executing identical code, but with different inputs from 20 different operators[5]. It is important that the instances of replication should be indexable in some way, and not be required to all have distinct non-indexed names.

Related to the need for replication, is a need for what may be described as polymorphic units. The word is taken from the terminology of object oriented systems, though the full power of polymorphic classes is perhaps not required here. What is needed is the possibility of creating units whose interfaces with the rest of the system are replicated, but whose implementations may be different. These should be capable of being substituted for each other in the system construction or during system execution. The purpose of this facility is to allow flexible replacement of software when hardware is changed in some way. The commonly mentioned example for this need is in connection with mode switches in fault tolerant systems, when recovery from an error situation following loss of some hardware may need a switch to a control strategy which is less expensive in processor cycles. However, the same facility would be needed in other situations. For example, a cash till in a banking system might be replaced by one of a different manufacture, providing the same facilities but requiring different algorithms for its control. Indexed sets of such units should be capable of being constructed from a mixture of the different versions.

We thus believe that the same capabilities must be provided for both distributed systems and non-shared memory parallel computers.

From this, one can derive a basic core of capabilities that must be provided in any system supporting distributed and/or parallel computation.

1. A mechanism to make data types visible to each of a multiplicity of processors so that arguments can be passed in remote calls.

2. A way of creating multiple instances of servers on the different processors that are not intended to be called remotely on different processors. Math packages and TEXT_IO are examples of this case.

3. A technique for Creating multiple instances of remotely callable units on different processors.

4. A means of assigning of remotely callable units to specific [logical] processors in the system.

5. A facility to replace remotely callable units by alternative versions with the same interfaces, but having different implementations.

In addition, certain forms of parallel operations have also been suggested as a basic requirement[1]. In particular, a capability for parallel creation of multiple instances of program entities, e.g., tasks, has been identified. This, in turn, would seem to require some form of broadcast communications capability.

We take these capabilities as the fundamental requirements for distributed and/or parallel Ada. The most basic problem that must be resolved is the identification of the sort (or sorts) of unit of distribution to be used and the rules/conventions that surround its, or their, use.

2.2 Underlying Considerations

Before delving into specific alternatives for distribution and parallel computation, it is useful to briefly review issues discovered in various experimental distributed Ada systems built over the past few years, e.g. [2], that must be addressed. In describing these issues, we do not assume any specific unit of distribution. Instead we introduce the term **par_unit** to identify the unit of distribution. We do not treat a **par_unit** as a prescribed language element at this point. Rather, we allow it to take on different characteristics and evaluate the consequences. In particular, we consider both task-like and package-like characteristics, as both tasks and packages have been suggested as the basis for distribution and parallelization. Also, in view of the discussion in the previous section, we keep in mind the need to replicate **par_units** in some way. Use of a type together with the allocator would provide a nice way to accomplish replication that is consistent with current mechanisms in the language. However, other forms of replication may be preferred if **par_units** are allowed at the library level.

We will consider the **par_unit** to have a specification and a body. The specification will present a remotely visible interface, and will include the remotely callable, accessible or visible entities in the **par_unit**.

Specific issues of concern are:

- *Avoid redundant state*: Maintaining consistency would be unreasonably difficult. This issue can arise if unrestricted replication of certain choices for a **par_unit** is permitted.

- *Avoid remote timed task entry calls*: As no system wide sense of time is defined, remote entry calls are not well defined [6].

- *Passing access variables as arguments in remote calls*: As presently defined, access variables have no meaning except on the machine upon which they received their values. While it is not difficult to create a record containing a machine (and possibly port) identifier as well as a local access value, this would either require a generalized implementation of operations on access variables or disciplined use of access variables, i.e., ensuring that they are only dereferenced on the machine on which they have meaning.

- *Avoid passing task objects as arguments in remote calls*: A task object evaluated on one machine has no meaning on a different machine.

- *Avoid hidden remote references*: This problem arises when instances of unrestricted task type definitions are created on machines remote from the one on which the type definition resides. The example below in which we take **par_unit** types to be task types illustrates the problem.

HIDDEN REMOTE REFERENCES

```
Site 1              Site 2

with B;             package B is
package A is        task type T is ···
    T1: B.T;            end T;
```

```
            ⋮                              ⋮
        end A;                         end B;
                                       package body B is
                                           X : INTEGER;
                                           task body T is
                                           ⋮
                                           begin
                                               X := ··· ;
                                           end T;
                                           ⋮
                                       end B;
```

Each instance of task type T will make a reference to the variable X. If the instance was created on a machine different from that on which the type definition resides, this reference will be remote, and will not be visible to the user. This problem also arises in the object oriented world if there are class variables (i.e. variables shared by all instances of a class) as well as instance variables, replicated with each instance. SmallTalk has them, and Eiffel allows only read only access to them, so they are really (usually remote) functions, while Dragoon has no class variables. **Par_units** should be constructed to avoid this problem. It is one of the most serious problems that must be addressed.

- *Data types defined within* **par_units**: If a **par_unit** is allowed to have types in its specification, two kinds of issues arise.
  - For each of the three kinds of operations on instances of the types defined within a **par_unit**, basic, implicit and user defined, one must decide whether to replicate the operation on each site using the type or make make remote calls to the operation. A compromise is to replicate the basic and implicit operations, and make user defined operations remotely callable. However, this means that the operations on a type are not treated consistently. Further, if they have side effects, then they all must be remotely called. If they are pure procedures or functions this compromise is workable.
  - The second issue arises when one considers creation of multiple instances of a **par_unit**. A "types within types" problem then arises. In particular, the types declared within different instances of the **par_unit** are presumably different since the instances are different. At a minimum, this, in turn, necessitates dynamic type checking, which creates a difficult implementation issue or use of type conversion.

    However, if Ada 9X tagged types are used as parameters in remote calls to a **par_unit** a problem arises with the distinctness of types in the multiple instances of included normal packages. This is illustrated in the example shown below[7]. In this case Pure and Remote Call Interface Packages of Ada 9X are used for distribution in a manner requiring replication of both Pure and normal packages "withed" by an RCI package. The problem arises because Ada 9X permits the passing of actual parameters that are extensions (defined in an included normal package) of the formal types given in the subprogram specification. When such a parameter is passed down to an instance of a normal package, the type extension defined in the receiving subprogram should, theoretically, be distinct from that defined in the instance creating the parameter, and a type mismatch will occur.

```
package PURE_PKG is
        pragma PURE;
        type ROOT_TYPE is tagged ...;
        - - Primitive dispatching operations ...
end PURE_PKG;

with PURE_PKG;
package RCI_PKG is
        pragma REMOTE_CALL_INTERFACE;
        procedure CLASS_WIDE_OP(CW_FORMAL
                : in PURE_PKG.ROOT_TYPE'CLASS);
        - - Note: class-wide nondispatching remote operation
end RCI_PKG;

with PURE_PKG;
package NORMAL_PKG is
        type SPECIFIC_TYPE is new PURE_PKG.ROOT_TYPE ...;
        - - Note: must be derived at the same scope level
        procedure P(ST_FORMAL: in SPECIFIC_TYPE);
end NORMAL_PKG;

with NORMAL_PKG;
package body RCI_PKG is
        procedure CLASS_WIDE_OP(CW_FORMAL : PURE_PKG.ROOT_TYPE'CLASS) i
        ...
        begin
                NORMAL_PKG.P(CW_FORMAL);
                ...
        end CLASS_WIDE_OP;
        ...
```

[7]This example is courtesy of Anthony Gargaro.

```
end RCI_PKG;

with NORMAL_PKG, RCI_PKG;
        - - Assume post-compilation partitioning creates this partition
        - - and it is different from the partition holding RCI_PKG.
procedure CLIENT_PTN is
        CLIENT_PTN.ACTUAL : NORMAL_PKG.SPECIFIC_TYPE := ...;
begin
        RCI_PKG.CLASS_WIDE_OP(CLIENT_PTN.ACTUAL);
        - - Remote call to RCI_PKG. Note that the parameter will be passed to the
        - - instance of NORMAL_PKG included with RCI_PKG, which is different from
        - - the instance included with CLIENT_PTN. If the types of different instances
        - - of NORMAL_PKG are different there is a type mismatch when RCI_PKG calls
        - - NORMAL_PKG.P.
end CLIENT_PTN;

- - Post-Compilation Partitioning - illustrative syntax:

PARTITION (PTN => 1, CLOSURE => (CLIENT_PTN))
PARTITION (PTN => 2, CLOSURE => (RCI_PKG))
- - Note: A distinct instance of NORMAL_PKG is included in each partition
```

- *Minimize task and program termination problems*: [2] describes task termination difficulties that can arise with unrestricted use of tasks.

3 Solution Concepts

In this section, we discuss alternative solution concepts for the basic capabilities identified earlier, and evaluate them with respect to the issues identified above.

3.1 Type Sharing

There has been fairly wide agreement on one method of achieving type sharing. Package-like units which have type declarations but no variable state are defined. They can then be replicated on each processor/memory system using the types they define, with the effect of having a single instance of the unit. Various forms of such units have been called **templates, publics, pure packages** or **constant state packages** by various investigators. For convenience, we will refer to units of this type as **type_units**.

The "types within types" problem can be avoided here by ensuring that, from the perspective of the program, there is semantically only a single instance of each defined **type_units**, and that its replication is only an operational optimization to reduce communication times. The difficulty that must be surmounted is the idea that a constructor function can determine where the state of a type instance is stored, and that the replicated units would place such storage on the machine on which they execute. The ensuing problems can be avoided by disallowing of access variables pointing to instances of types defined within the **type_unit** as parameters in remote procedure calls.

One important issue that has been treated differently in the variations of this approach, though, is the presence or absence of task types. They are disallowed in **publics**[7] but allowed in Ada 9X **pure** packages. The possible advantages of including them are discussed more conveniently later. However, if they are permitted, it is necessary to disallow passing task objects as parameters on remote calls, and this would seem to require more complex checking than prohibiting them altogether.

It should be noted that the August 1991 version of Ada 9x also allows type sharing through placement of type declarations in the specification of a Remote Call Interface (RCI) package. This is needed to have private types. Furthermore, RCI packages may have state.

3.2 Assignment of Callable Units to Processors

There are several considerations in assignment of callable units to processors:

- Configuration of the program, i.e., the logical division of the program into parts to be executed on different machines.
- Automatic management of communication routines. That is, calls to remote subprograms must be replaced with communication stubs, and the communication routines must automatically be supplied.
- Assignment of units to specific hardware.

A number of different approaches have been taken in addressing these problems. The details of the different approaches depend, to some extent, on the specific kind of unit of distribution with which they are dealing. However, certain general characteristics are more or less independent of the unit of distribution. These alternative approaches are:

- Develop the program for a virtual computer and use a post-processing tool to assign different parts of the program to different [logical] machines[8].
- Use pragmas to identify the units to be distributed and the [logical] processors to which they are assigned[9].
- Use configuration tool that can specify the [logical] distribution of the program as part of the high level program design[10].
- Incorporate explicit, user written, [logical] assignment statements within the program itself[7].

- Embed algorithms for automatically performing the [logical] assignment in the compiler.
- Perform logical machine to physical machine assignment at the time the program is loaded[9].
- DRAGOON takes the object oriented view to its logical conclusion in which the complete program is an object configured from others in a consistent way.

Combinations of these can be used. As several methods of accomplishing the assignment to specific machines exist, this requirement will not be considered further here. Any of several methods will suffice.

3.3 "Local" Replication of Units

As noted above, it is useful to be able to create local instances of various servers, such as TEXT_IO and math packages, on the individual processors of a distributed or parallel computer system. In the case we consider in this section, we assume there are no remote references to such servers. The need for replication in this case differs from that for **type_units** in that the replicated units under discussion here may have variable state. We use the term "local replication" to describe the process of creating instances of such server units on the processors on which they are needed. Since locally replicated units are not remotely callable, they must appear in the context clauses of remotely callable **par_units**. Library packages and subprograms are appropriate units for local replication.

The ideas of local replication have appeared in both AdaPT [7] and in Ada 9X (per-partition packages) [1]. It is generally agreed that: 1) each instance of the replicated unit should be a distinct copy of the defined unit, 2) each instance of state associated with a replicated unit should be separate and distinct from all others, and, 3) there should be no attempt to maintain state consistency. These effects are obtained in object oriented languages (e.g. DRAGOON) by having an instance variable of the shared class instantiated in each instance of the class. If they are shared, then the instance variables are assigned a common value, which, of course, creates a consistency problem in the distributed case.

The matter of types within locally replicated units is less clear. Ostensibly, there is the "types within types" problem identified earlier. Were it not for the tagged types proposed for Ada 9X, however, the problem would be moot. Except for tagged types, there can be no mixing of the different instances of the data types defined in the unit since locally replicated units cannot be referenced from without the processor on which they reside and scope rules thus prevent passing objects of types in one instance to another.

There is, however, the types within types issue when tagged types are considered. If the types within different instances of a locally replicated unit (on different machines) are distinct, then the problem illustrated earlier with passing parameters of a tagged type arises. The August 1991 Ada 9X Mapping Specification resolves this problem by decreeing that the types within different instances of per-partition packages are the same (while state within these same different instances is distinct).

This, we believe, is inconsistent with the notion that the replicated instances are distinct, and we recommend a different approach. We recommend that either the passing of tagged types as parameters in remote procedure calls be disallowed, or that only tagged types declared in a **type_unit** be allowed as parameters (i.e., no 'CLASS formal types), and that the copy of the actual parameter have all extensions to the formal type "stripped off." Either choice prevents an object of a tagged type extended from a type declared in a **type_unit** from reaching a different instance of the same base unit.

With our suggested approach, there is an interpretation of local replication that fits nicely into the Ada philosophy. Locally replicated units are essentially instances of anonymous package or subprogram types. There is no need to develop any new typing mechanism for specifying "locally replicable" units, however. If they are included in the context clauses of the remotely callable **par_units** that need them, they can be automatically included where needed as linked executable objects are produced for each machine of the system.

3.4 Remotely Callable Replication

In this case, it is desirable to be able to name the instances of replication, create structures containing them (or pointers to them) and, at least locally, pass them (or pointers to them) as parameters. It is also essential that the units have, or be able to create, one or more threads of control.

There have been a number of proposals for resolving this issue, though there is as yet no widely accepted agreement on the best approach. Package types have been suggested for this purpose [11, 2]. Partitions and nodes were suggested in AdaPT [7]. DRAGOON uses class instances (which really influenced the proposal for AdaPT). The August, 1991 Ada 9X Mapping Specification [1] included a parameter in task types so that each task could be given an identity as it is created; it was intended that compilers be allowed to distribute an array of tasks across a set of parallel processors. Additionally, the draft Ada 9X document proposes Active Partitions, accessible via Remote Call Interface (RCIs) packages for distributed processing. Duplication of RCIs (not presently allowed in Ada 9x) would be an additional form of replication. And, most recently, [12] has suggested an abstract data type (ADT) model in which replication of a remotely callable suitably formed **type_unit** permits a realization of the AdaPT model within Ada83 or Ada 9X.

In order to not pre-judge a solution, we will again use the term **par_unit** to name the unit whose instances are to be replicated on different processors. We will consider different possible characteristics of **par_units** that would make them useful for distribution and parallel computation. At times, we will consider the consequences of having them resemble tasks. At other times, we will consider the consequences of having them resemble packages, or the partitions of AdaPT [7]. At all times, though, we will consider that **par_units** must actually be, or represent, types.

There are at least three different bases upon which **par_units** can be developed:

1. **par_unit** types appear at the library level.
2. **par_unit** types are embedded within the definition of some other unit, as are **tasks** and.
3. **par_units** are are implemented as Abstract Data Types (ADTs).

For each of these choices, it is necessary to consider the conditions that further refine the definition of a **par_unit**.

3.4.1 Library Level Par_units

In this case, **par_unit** definitions resemble package or subprogram definitions constrained in appropriate ways (to be discussed below), with the major modification that we wish to create multiple instances of them. Since the normal Ada mechanism for creating instances of things is based upon types, we should thus think of a **par_unit** definition as a type definition. Even if we do not formally introduce a typing syntax into the language, but instead use some form of configuration control to achieve multiple instances of **par_units**, it is useful to recognize that conceptually we are dealing with types.

The major issues to be considered in this case are:

- Types required by the **par_unit** itself
- Context of the **par_unit**
- State of the **par_unit**
- Creation and naming of the instances

If one is willing to disallow private types in the specification of a library level **par_unit**, one can avoid allowing types in a **par_unit** specification altogether by placing the visible types needed in some form of **type_unit**. The prohibition of data types in **par_unit** specifications eliminates the types within types problem. The prohibition of task types eliminates part of the hidden remote reference problem. To eliminate other forms of hidden remote references, generic definitions within the specification of a **par_unit** should also be prohibited. Task declarations should be disallowed to prevent remote timed entry calls. The effect of normal entry calls is easily obtained by having visible procedures make calls to tasks defined in the body of a **par_unit**.

The context clause of a **par_unit** should only reference whatever form of **type_unit** is adopted. Further discussion on possible conditions for **par_unit** may be found under the heading of partitions in [7] and Segment Interface Packages in [13]. The body of the **par_unit**, however, may include normal packages and subprograms in its context.

Each instance of a **par_unit** should be be distinct, i.e., the state in one instance is distinct from that in another. Any normal packages or subprograms included by the body of a **par_unit** should be locally replicated, with a distinct copy being included in each instance of the **par_unit**. This creation of distinct instances of **par_unit** state avoids the redundant state problem.

While there is no fundamental reason for disallowing state in the specification of a **par_unit**, it is convenient to allow state only in the body. State in the body is readily accessible via subprogram calls, and the absence of state in the specification simplifies the translation process.

The creation of multiple instances of library level **par_units** could be accomplished by several different methods. First, as in [7], the language syntax could be extended to syntactically allow **par_units** as types, access variables to **par_units** and use of the allocator or declarations to create instances. The following sample of code illustrates the idea.

Illustration A

```
par_unit type B is ...
    type T is ...
    ⋮
end B;
```

In this case, B would be a library unit, similar in many respects to a generic package. Instances of it could be created via the allocator, as for example,

```
type P is access B;
P1: P := new B;
```

The configuration of P1 to a specific processor could be handled by adding a logical processor ID to the allocation above, with a binding to a specific processor occurring at link or load time. This might be the "purest" manner from a language theory perspective, and is similar to what was done in AdaPT [7]

Alternatively, one could simply treat **par_units** syntactically much like packages. That is, one could simply declare:

Illustration B

```
par_unit B is ...
    type T is ...
    ⋮
end B;
```

Specific instances could be created in any of three ways, two of which are outside the language:

1. Make B a generic library unit and use generic instantiation.
2. Use a pragma to specify the [logical] sites on which instances of B are to be placed.
3. Use an external configuration tool to specify the sites on which instances of B are to be placed.

In the second two cases, an additional mechanism would have to be developed to specify, in a remote call, which instance of B was being called. While we are unaware of an implementation doing this, we believe it is possible. One possible method will be outlined later.

If one must be able to use private types with **par_units**, one must, as Ada 9X does with Remote Call Interface Packages, allow type definitions within **par_units**. In this case, the "types within types" problem arises. Strictly speaking, the types within different instances of a **par_unit** should be distinct. The types within types problem motivating this can be illustrated as shown below using the syntactic type model of **par_units** given in Illustration A above:

TYPE WITHIN TYPE PROBLEM

```
type P is access B;
P1, P2: P : = new B;
P3 : P;
V1 : P1.T;
SW : BOOLEAN := IN_FUNC;
:
begin
    if SW then P3 := P1 else P3 := P2; end if,

    declare
        V3 : P3.T;
    begin
        if V1 = V3 then - - legal if P3 = P1, illegal if P3 = P2
        :
    end;
end;
```

A similar problem can occur with the second model of **par_units** using generic instantiation. In both of these cases, use of type conversion could resolve the problem, though it might be required extensively.

Alternatively, instances of B could be created using pragmas or a configuration tool with a **par_unit** declaration as in Illustration B. There would then be no syntactic mechanism for creating pointers to the different instances and hence no way to declare objects such as V1 above corresponding to a specific instance of the **par_unit**. That is, one could only declare a variable as X: B.T; there would be no way to specify a particular instance of B from whose instance of the type would be used. The specific example types within types problem given above could not then occur. However, there is still a difficulty.

While commonality of a type across all instances of a **par_unit** might be exactly what a programmer wants, it does create an inconsistency. Some entities resulting from declarations in a **par_unit** (variable state) are distinct in different instances of the **par_unit**, while others (types) are the same.

There is a perspective that resolves the disparity, however. [7] introduces the notion of "conformance." Conformant units have the same specification, but different bodies. Instances of **par_unit** created via pragmas or configuration tools could be considered conformant **par_units** having the same specification, in which case allowing types within the specification of a **par_unit** would cause no problems.

One might wonder if the notion of conformance might not also be used with local replication, and thus eliminate the tagged type problem described earlier. The notion of conformance was not defined for entities having state in their specifications as normal packages may. However, if the general idea of identical specifications of conformant entities were carried through, it would be inappropriate since conformant instances of normal packages having state in their specifications would be expected to maintain state consistency.

3.4.2 Par_units Embedded Within Some Context

The alternative to defining **par_unit** types at the library level is have them defined within some other context, much as task types are in Ada '83. In this case, the central issue is the relation between the **par_unit** and its context when instances of the **par_unit** are created. The previous illustration of the hidden remote access problem portrays the problem in its simplest form. More generally, one must be concerned about the entire transitive closure of the context of the **par_unit**, including all subprograms, variables, tasks, etc. in the context.

There are four basic choices, one of which *must* be chosen for each (or all) kind(s) of referencable items (variables, subprograms, tasks, etc.) in the context of the **par_unit**.

1. *There is but a single instance of each object in* **par_unit** *context*:
   This is exactly the hidden remote reference situation described earlier for tasks. It would be virtually impossible for a programmer to predict even gross characteristics about the timing behavior of his/her program. This is almost assuredly unacceptable from a user perspective. Furthermore, implementation is likely to be difficult since it is not known until the time an instance of the **par_unit** is

created whether a reference is local or remote [9]. In the opinion of the authors, this alternative must be rejected.

2. *Replicate the context of a* **par_unit** *for each instance*:
   On the surface, this seems similar to the local replication of context discussed above for library level **par_units**. However, there are very important differences that make the replication much more difficult in this situation, if not an illusion in the general case.

   The context that must be replicated includes not only the state in the outer scope of the **par_unit** and the transitive closure of applicable **with** clauses, but callable entities as well. If callable entities were not included, then the calls to callable entities from instances of the **par_unit** would be remote and the called entity would either not know which instance of replicated state to use, or would require some complex form of state passing.

   Further, entities in the context of the **par_unit** can themselves be directly called from without the **par_unit**. Indeed, in some cases instances of the **par_unit** may only come into existence as the result of such an external call, as, for example, if the **par_unit** is declared within a procedure, P. If P must be replicated because it is in the outer scope of the **par_unit** it declares, then there are multiple instances of P and which one must be called to create the instances of the **par_unit**? The whole process degenerates.

   Moreover, the replication would have to take place at dynamically run-time. Aside from being difficult to do, the process is likely to be slow, exactly the opposite of what one seeks to do. Finally, for embedded **par_units**, the replication is likely to be quite counter-intuitive for programmers.

   Without some further restrictions, this approach does not appear feasible.

3. *Disallow the creation of objects in the context of a* **par_unit**:
   In particular, this means that no variables or subprograms may be declared in any unit in the context of the **par_unit**. For **par_units** defined in arbitrary places in a program, e.g., within a procedure within a package, etc., this is very restrictive, and requires extensive compile-time checking unless there are restrictions on where **par_units** may appear in a program.

4. *Disallow reference to objects in the context of a* **par_unit**:

   While this is feasible from an implementation perspective, it is inconsistent with current scoping rules.

In the opinion of the authors, none of these choices is acceptable for arbitrarily placed **par_units**. That is, arbitrarily placed **par_units**, such as tasks, embedded within an Ada program are an unacceptable way in which to achieve distribution or parallel computation.

However, if one places restrictions on where **par_units** may be declared, embedded **par_units** become more feasible. In particular, consider the following restrictions on embedded **par_units**:

1. **Par_units** may be declared only in library level units.

2. The unit declaring a **par_unit** may have no variable state.

3. The unit declaring may include only **type_units** in its context.

In this case, there is no variable state to replicate, and the checking that these conditions have been satisfied is relatively easy to perform. Further, with only a few other relatively minor restrictions, the unit containing the the **par_unit** definition can itself be replicated on any machines that might need it, facilitating subsequent implementation. This is close to the situation with Pure packages in the August, 1991 Mapping Specification for Ada 9X.

The process of creating a large number of **par_unit** instances and placing them on different processors in the system is a major concern. It is highly desirable to be able to do this in some parallel fashion. As this involves the same underlying mechanism as some desirable operations in the library level **par_unit** case, the discussion of **par_unit** creation is deferred to the section on parallel operations.

3.4.3 The Abstract Data Type View

The Abstract Data Type (ADT) view of **par_units** essentially reduces them to **type_units**. The basic idea is to collect all state in an instance of a **par_unit** and the transitive closure of its context clauses into one, possibly large, record, a type for which is included in the **par_unit** declaration. An instance of this record then provides the state for the **par_unit**. Creation of an instance of the **par_unit**, then, is simply the creation of an instance of the state record type and with the return of a pointer to it.

Since instances are required on different machines, the pointer concept must be extended to a record that includes at least a machine id in addition to the local reference to the instance of the state. The following code segment illustrates the definition of an ADT **par_unit**. See [7] for a complete description of this process, which is actually a bit more complex than illustrated here because the AdaPT model includes both partition and node types.

```
type MACH_ID is ... - - an enumerated type listing machines
package AN_ADT_PAR_UNIT is
    type STATE_PTR is private;
        type STATE is
            MACH: MACH_ID;
```

```
                REF: STATE_PTR;
            end record;
        procedure PAR_UNIT_PROC(...; S: STATE);
        procedure CREATE(S: in out STATE);
    private
        type ADT_STATE; - - this is the composite state type.
        type STATE_PTR is access ADT_STATE;
    end AN_ADT_PAR_UNIT;
```

Since the **par_unit** now has no variable state, it may be freely replicated on all units needing an instance of it. That can be done by a configuration process outside the definition of the language. Creation on an instance of AN_ADT_PAR_UNIT requires a remote call to the CREATE procedure on the machine on which the unit is desired.

While the ADT model reduces the AdaPT model to standard Ada or Ada 9X using stateless packages, distribution or parallel operation can only be achieved if these units may be replicated on different processors, and specific instances referenced. The ADTs appear similar to **type_units** such as Pure packages. They differ, however, in that they must be remotely referencable. Replication is not now a simple question of optimization of communication, as all references to **type_units** such as Pure packages are to local copies of these units. Though one could use a replicated stateful unit (such as replicated RCIs), the whole idea of the ADT model was to eliminate state in the specification.

One can conceive of a clever way to use **type_units** such as Pure packages to implement ADTs, however, if one builds upon the idea of an intelligent communication system. Subprograms in a Pure package could have an additional parameter of type STATE which is defaulted to a value having some special "self" code for the MACH component. Communication stubs could be provided that would check the parameter of type STATE and determine whether a local or remote call was being made. Because of the default for this parameter, local calls would not even need to include it.

Invocation of an operation of the par_type, e.g., PAR_UNIT_PROC, requires passing a pointer to the state object as one of the parameters to the operation. The communication system must be able to examine the MACH field of the S parameter and direct the call to the proper machine. This is quite possible if all ADTs are constructed in the same manner, and this, in turn, is quite feasible since one may use a pre-processor to automatically convert something like AdaPT into an ADT form.

The process of "flattening" a **par_unit** and its context to obtain the composite state buffer is difficult to do by hand for large systems, but could easily be performed by a suitable pre-processor.

3.5 Parallel Operations

A degree of parallelism is achieved by placing instances of **par_units** on different processors in the system, each with its own thread of control. Communication among them, however, then becomes an obvious potential bottleneck. Additional efficiency can be obtained if part or all of the communication can be parallelized. There are two kinds of operations in which attempting to achieve parallel communication is likely to be effective. The first occurs when one program segment must perform the same call to a set of **par_units** on different processors, such as might be done to initiate some parallel operation. The second is the creation of a set of instances of some **par_type**.

Parallel communications can only be used effectively when the same message can be broadcast to a set of processors. Effectiveness can be enhanced if no return is expected. Furthermore, the mechanisms involved are much more complex if multiple returns are received to a single broadcast. We thus consider first the case in which there is no return.

3.5.1 Parallel Communication With No Return

Case 1: Communication

In the case of parallel calls to a set of **par_units**, the lack of return messages implies only **in** parameters. Such parallel calls can be handled by an extension of the ideas used in the ADT example together with placing a degree of intelligence in the communication system. We begin be defining a type for a set of MACH_IDs.

```
type MACH_SET is            - - a set of MACH_IDs.
```

Then, if there is a procedure E in a **par_unit** PAR, its specification would appear as follows:

```
par_unit type PAR is
    procedure E("parameters", MS: MACH_SET);
        . . .
end PAR;
```

The code for PAR would be replicated on the processors in the system in some way (the method is not of concern here). The compiler would be required to recognize that references to PAR.E were remote and substitute communication stubs in place of the actual procedure on the processor making the call. This kind of detection and substitution has already been implemented in at least one of the trial implementations of the Ada 9X distribution mechanism.

The communication subsystem must be intelligent enough to recognize that a call to PAR.E with the parameter MS is actually a parallel call. It should broadcast the call, and, since there are no parameters to be returned, immediately return to the calling thread of control. The receiving part of the communication subsystem must also be intelligent enough to recognize whether or not a received call is intended for the machine on which it resides, ignoring it if it is not, and completing the call if it is. The receiving communication subsystem has the information to make this decision since it can compare its ID with those contained in the parameter MS. PAR.E itself would not actually use the parameter MS.

MACH_ID was only used for illustrative purposes to show the fundamental idea. In general, it would be a record containing whatever set of identifiers are necessary to uniquely identify the called unit, which might include, for example, some form of instance number local to the machine on which the instance resides as well as the machine id.

Case 2: Process Creation

Parallel creation of **par_unit** instances might proceed by parallel calls to either an explicit CREATE routine similar to the ADT example or an allocator in the run time systems of the units on which the instances are to be created. The difference between these calls and those discussed above is that the invoking program segment will almost always need to have some form of identifier (value) for each **par_unit** instance created so that it can reference the instance in the future.

If no return from the called units is allowed, then the invoking program must supply the identifier (value). This means that the receiving unit must store the identifier so that it can match incoming calls against it to determine if the created unit is being called. This, in turn, means that the unit which creates the **par_unit** instances must have variable state.

Since matters of state and types are key factors in determining what constraints must apply to **par_units**, this conclusion is significant. For example, letting **par_units** be task types, it means that this form of parallel creation cannot be used to create remote instances of tasks in Ada 9X, i.e., it cannot be used for tasks whose types are declared in Pure packages in Ada 9X because Pure packages may not have variable state. Further, it could not be used with creation of tasks in Remote Call Interface packages because RCI packages may not have task types in their specifications.

3.5.2 Parallel Communication With Return

One can conceive of a system that could handle return values from a parallel call. The returns, of course, would have to be received serially, though the call could be parallel. In concept, a parallel call with return could be constructed as follows:

- The argument list to the called subprogram must include an out parameter which is a SET of outputs. Each output variable must include a component to hold the identifier of the instance of the called subprogram returning the parameter. As in the case of no returned variables, the argument list must also include a set, MS, of called instances of the subprogram.

- Each instance of the subprogram provides an out value which is a singleton set. This value must include a component that identifies the instance of the subprogram providing the value.

- The client communication stub must await a return from each of the instances named in the argument set MS, and place the result in the output set. It returns to the caller only when a return value has been received from each called instance of the subprogram.

Whether the called subprogram instances are user defined or a system allocator is used is irrelevant. While such a scheme can be made to work correctly, it is obviously less efficient than the no return case since there must be a serialized return.

A parallel communication scheme such as this with serialized return could, however, be used to create instances of tasks from Pure packages in Ada 9X. It eliminates one half of the serialized communication that could occur in the creation of a set of instances of tasks from task types.

4 Distributed and Parallel Computation in Ada 9X

The August, 1991 draft Mapping Specification addressed distribution and parallel computation only briefly. Nevertheless, it does provide a base upon which a limited level of distributed and parallel systems can be built. We believe that some minor modifications to the MS will significantly enhance its utility for programming distributed and parallel systems. We present these recommendations in two levels, graduated according to the degree of modification and enhancements involved.

4.1 Level of Least Change

At the level of least change we would suggest the following:

1. Allow active partitions to be replicated.
2. Explicitly recognize the instances of Remote Call Interface Packages as being conformant packages.
3. Only allow parallel task creation (if any is allowed at all) from task types specified in Pure packages.
4. Either disallow the passing of tagged records as parameters in RCI calls or disallow the use of 'CLASS tagged parameters, as suggested earlier.
5. Disallow the passing of tasks created from task types in the specification of Pure packages as parameters in subprogram or entry calls.
6. Allow only a single Remote Call Interface (RCI) Packages per active partition.

The most important item is the ability to replicate remotely callable entities. This facilitates use of the ADT model for distribution (as an alternative to the use of Pure packages as described above), management of fault tolerance through the ability to create backup units, and the management of parallel processing with minimal change to the language. Each instance of an active partition must include not only a separate copy of the RCI package within it, but a locally replicated copy of everything, except for other RCI packages, in the transitive closure of the context clauses on both the specification and body of the RCI. Allowing only one RCI package in a partition is not essential, but will be a convenience in addressing different instances of an active partition.

Implementation designed configuration tools and utility procedures can be adjoined to the implementation defined communication facilities to effect the distribution and communication among the RCI packages instances.

As the replication of RCI packages creates a "types within types" issue, the matter must be resolved in some way. The simplest and most natural way is to recognize the instances of RCI packages as conformant instances. With this decision, nothing special need be done. The types in the specification of an RCI would then be the same for all instances.

The restriction on tagged types as parameters in RCI calls eliminates the problem of type matching in normal packages included in multiple partitions.

As pointed out above, an attempt to dynamically create parallel instances of tasks from task types defined at arbitrary levels in a program is fraught with difficulties. The worst difficulties can be avoided if the creation of parallel tasks is limited to tasks created from task types appearing in the specification of Pure packages. Pure packages have no variable state, and can be replicated on each processor in the system. Thus, the context of the tasks created can be easily made locally available to them. Various implementation provided mechanisms can then be developed for creating and referencing parallel tasks, such as the one outlined above.

Passing task objects across machine boundaries will cause considerable implementation difficulty as well as making it difficult for a programmer to anticipate the behavior of his/her program, and should thus be disallowed. As the only task types visible across machine boundaries are those in Pure packages, disallowing the appearance of tasks of these types in subprogram or entry calls is sufficient to prevent the problem.

4.2 Level of Moderate Change

The items recommended at this level could easily be considered independently. They are grouped together simply for convenience. At the level of moderate change the following suggestions are made:

1. Disallow the presence of non-private types in the specification of RCI packages.

2. Disallow the presence of task types in the specification of Pure packages.

3. Add a pragma REMOTE_CALL_PURE to designate Pure packages for which communication stubs are to be generated.

There is really no need to include non-private types in the specification of RCI package's, as they can be provided through Pure packages. If they are eliminated, the issue of "types within types" goes away and does not have to be considered at all.

Similarly, it is possible to achieve parallelism without having task types in the specification of Pure packages through use of replicated RCI packages. As described earlier, a call (perhaps broadcast) needs to be made to a remote unit of some form to create a parallel instances of a task. Whether this task object is created from a Pure package or from a task type within the body of an RCI is immaterial. Calls to a task within an RCI body can easily be handled via a procedure declared in the specification of the RCI. There is thus very little difference from a programmer's perspective which method is used. The implementation of parallel tasks via Pure packages is redundant, and could be deleted as it complicates the implementation.

It should be noted that with the use of RCI's, tasks could also be declared directly in the body of the RCI package and and elaborated when the RCI is. In this case, the programmer could simply call the task (indirectly via a procedure) to start it, and would not have to create it.

The omission of task types from the specification of Pure packages eliminates the need for any concern about the programmer trying to pass task objects as parameters.

It is not strictly necessary to add the pragma, but it might be more consistent to do this in the distribution annex rather than let implementors choose their own name for it.

5 Summary and Conclusions

The major language issues impacting distributed and parallel programming have been reviewed, and some principles upon which distributed/parallel language systems should be build suggested. Based upon these, alternative language concepts for distributed/parallel programming have been analyzed.

The most fundamental conclusion is that there is a need for replication of library level units for both distributed and parallel language systems, and that this, in turn, is most properly viewed from the perspective of a typing model. It is not necessary, however, to introduce syntax into the language to accomplish this as external configuration tools, in conjunction with appropriate communication capabilities, can achieve the necessary effect. Replication of library level units would facilitate a general Abstract Data Type model of distribution that permits implementation of powerful distributed programming mechanisms, as well as enabling distributed and parallel programming in general.

It has been previously suggested that it is useful in some circumstances to have a typed language construct embedded within program units (i.e., something task-like) whose instances can be distributed. Our analysis reveals that considerable care must be taken in developing the conditions under which this is possible. We have shown that only in limited cases is it feasible. though in those cases, it may be quite useful.

In addition, we have sketched a communications functionality that would augment language mechanisms to support parallel and distributed operations.

Finally, we have suggested a number of specific minor changes to enhance the suitability of Ada 9X for distributed and parallel programming, and avoid major difficulties that have been encountered in previous distributed Ada implementations.

References

[1] Office of the Under Secretary of Defense for Acquisition, Washington, D.C. *Ada 9X Project Report, Ada 9X Mapping Document Volume II, Mapping Specification*, August 1991.

[2] R. Volz, T. Mudge, G. Buzzard, and P. Krishnan. Translation and execution of distributed Ada programs: Is it still Ada? *IEEE Transactions on Software, Special Issues on Ada*, 15(3):281–292, March 1989.

[3] Richard A. Volz. Virtual nodes and units of distribution for distributed Ada. In *Ada Letters Special Edition, Vol X, NO 4 - 3RD International Workshop on Real-Time Ada Issues*, 1990.

[4] Raymond Scott Waldrop. Distribution of Ada using adapt in a heterogeneous environment. Master's thesis, Department of Computer Science, Texas A& M University, 1991.

[5] R. S. Waldrop, R. A. Volz, S. J. Goldsack, and A. A. Holzbacher-Valero. Programming in a proposed 9X distributed Ada, May 1991. Status report, subcontract #074 cooperative agreement NCC-9-16.

[6] R.A. Volz and T.N. Mudge. Timing issues in the distributed execution of Ada programs. *IEEE Trans on Computer for publication in special issue on Parallel and Distributed Processing*, C-36(4):449–459, April 1987.

[7] A. B. Gargaro, S. J. Goldsack, R. A. Volz, and A. J. Wellings. A proposal to support reliable distributed systems in Ada 9x. Technical report, Texas A&M University, 1990.

[8] Rakesh Jha, J. Michael Kamrad II, and Dennis T. Cornhill. Ada program partitioning language: a notation for distributing Ada programs. *IEEE Transactions on Software Engineering*, 15(3):271–280, March 1989.

[9] R. A. Volz, P. Krishnan, and R. Theriault. Distributed Ada: case study. *Information and Software Technology*, 33(4):292–300, May 1991.

[10] Ron Theriault. *Telesoft distributed Ada configuration tool.* Department of Computer Science, Texas A&M University and Telesoft Corporation, December 1991.

[11] W.H. Jessop. Ada packages and distributed systems. *SIGPLAN Notices*, February 1982.

[12] A. A. Holzbacher-Valero, S. J. Goldsack, R. A. Volz, and R. S. Waldrop. Transforming AdaPT to Ada. August 1991. Status report, subcontract #074 cooperative agreement NCC-9-16.

[13] Richard Volz, Ron Theriault, Gary Smith, and Amanda Mayo. *Telesoft distributed Ada configuration tool, distributed Ada language conventions.* Department of Computer Science, Texas A&M University, December 1991.