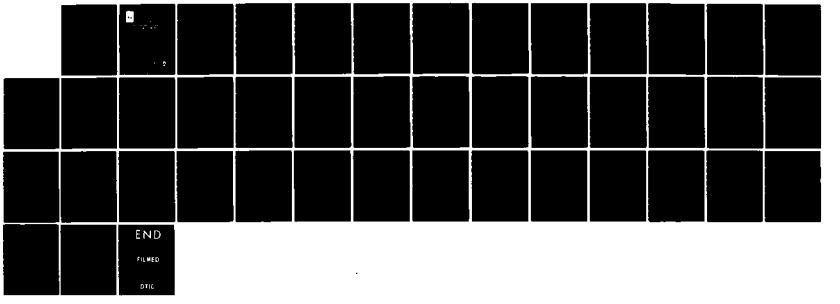
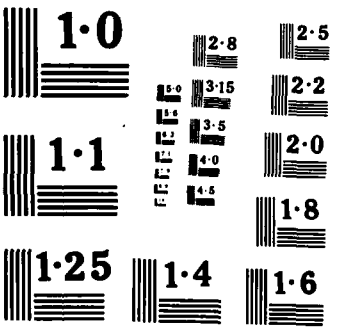
AD-A152 665 AUTOMATIC DISTRIBUTION OF PROGRAMS IN MASCOT AND ADA (TRADEMARK) ENVIRONM..(U) ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) G FICKENSCHER NOV 84 1/1

UNCLASSIFIED RSRE-MEMO-3696 DRIC-BR-94681 F/G 9/1 NL

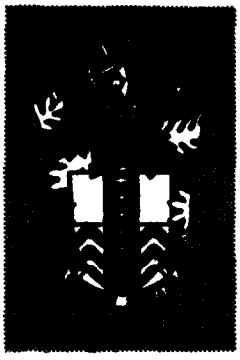



1·0
1·1
1·25
2·8
3·15
3·5
4·0
4·5
1·4
2·5
2·2
2·0
1·8
1·6

UNLIMITED

BR94681

(4)



RSRE
MEMORANDUM No. 3696

ROYAL SIGNALS & RADAR ESTABLISHMENT

AUTOMATIC DISTRIBUTION OF PROGRAMS IN MASCOT AND Ada ENVIRONMENTS - A FEASIBILITY STUDY

Author: Gustav Fickenscher

AD-A152 665

RSRE MEMORANDUM No. 3696

DTIC FILE COPY

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
RSRE MALVERN,
WORCS.

DTIC
ELECTE
S APR 23 1985 D
E

UNLIMITED

85 4 22 086

UNLIMITED

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 3696

TITLE: AUTOMATIC DISTRIBUTION OF PROGRAMS IN MASCOT AND Ada * ENVIRONMENTS - A FEASIBILITY STUDY

Author: Gustav Fickenscher

Date: November 1984

SUMMARY

Recent years have seen a steady increase of computer systems based on distributed hardware. This is made possible by the reduction of hardware costs and increases in the power of hardware components. More sophisticated software systems are the result. To reduce the costs for software development it would be advantageous to reuse software systems in different target environments. In the case of distributed systems this means that software may be distributed differently in different environments. It would simplify matters if the software could be distributed automatically. In this paper the feasibility of automatic distribution of Ada programs and MASCOT-like programs is investigated. The impacts of the distribution on the runtime environment and communication system are outlined.

Accession For
NTIS GRA&I [X]
DTIC TAB [ ]
Unannounced [ ]
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

DDC QUALITY INSPECTED 1

This memorandum is for advance information. It is not necessarily to be regarded as a final or official statement by Procurement Executive, Ministry of Defence

Copyright
C
Controller HMSO London
1984

---

* Ada is a registered trademark of the US Government, Ada Joint Progress Office.

CONTENTS

1. Introduction 1
2. The Hardware Environment 2
   2.1 Range of Distributed Systems 2
   2.2 Essential Hardware Elements 3
   2.3 Description Method 4
3. The Distribution of Application Programs 7
   3.1 Distributable Parts of MASCOT-like Programs 7
      3.1.1 Inter-Communication Data Areas 7
      3.1.2 Activities 8
      3.1.3 Subsystems 9
      3.1.4 Library Routines 10
   3.2 Distributable Parts of Ada Programs 10
      3.2.1 Visibility and Scope of Declarations 10
         3.2.1.1 Object Declarations 11
         3.2.1.2 Type Declarations 12
         3.2.1.3 Block Statements 12
         3.2.1.4 Subprograms 13
         3.2.1.5 Packages 14
         3.2.1.6 Generic Declarations 16
         3.2.1.7 Generic Instantiations 16
         3.2.1.8 Tasks 16
         3.2.1.9 Scope Hierarchy and Relations between Scopes 17
      3.2.2 Subunits 19
      3.2.3 Library Units 19
      3.2.4 Dynamic Creation of Task Objects 20
      3.2.5 Input-Output 21
   3.3 The Distribution Process 21
      3.3.1 Requirements and Strategies 21
      3.3.2 MASCOT-like Programs 22
      3.3.3 Ada Programs 23
      3.3.4 MASCOT/Ada Programs 23
4. Communication 24
   4.1 Communication Mechanisms 24
      4.1.1 Message Passing 24
      4.1.2 Remote Invocation 25
      4.1.3 Paired Input/Output Statements 25
   4.2 Communication in MASCOT-like Systems 26
      4.2.1 Invocation of Access Procedures 26
      4.2.2 Problems of Remote Invocation of Access Procedures 27
         4.2.2.1 Parameters of Access Procedures 27
         4.2.2.2 Communication Problems 28
         4.2.2.3 MASCOT Primitives 29
      4.2.3 Solution for MASCOT-like Systems 30
   4.3 Communication in Ada Systems 30
      4.3.1 Global Objects 30
      4.3.2 Subprogram Calls 31
      4.3.3 Entry Calls 31
   4.4 The Communication Protocol 32
5. The Runtime Environment and Associated Tools 33
   5.1 Ada Programs 33
   5.2 MASCOT-like Programs 34
6. Conclusion 36

7. Acknowledgements 37
8. References 37

1. Introduction

Recent years have seen a steady increase of computer systems based on distributed hardware. Such systems offer greater availability and greater reliability because of built-in hardware and possibly software redundancies.

The software involved is mainly tailored to the particular system. Therefore hardware changes normally imply major changes, eventually even redesigns of software components, or, even worse, a new development of the whole software. Nowadays, it is acknowledged by most people in the computer business, that the software system is the more expensive part of a computer system. The application system should be as hardware independent as possible and its design should not be constrained by target hardware but should only be based on software criteria. The final distribution of the application system onto the various hardware components should more or less be performed automatically. Such an approach would allow hardware changes leaving the software system almost unaffected. Ideally, the application system should not have any knowledge of the hardware it is running on.

If design and implementation of software are to be made independent of the target hardware, some questions need to be resolved leading to an automatic distribution process:

- In what way is a distributed target environment to be described?
- In what way can an application system be partitioned and how can the resulting parts be distributed?
- How is an efficient means of communication achieved between the distributed parts?
- To what extent are compilers, linkage editors, and loaders affected?
- What are the requirements for an underlying runtime system?

Ideally these questions should be solved independently of particular design methodologies and implementation languages, because they are of general interest. However, this paper restricts itself to MASCOT [2] and Ada [1]. The clear design guidelines of MASCOT ease the distribution process. Ada, on the other hand, seems to complicate the problem with its special tasking and package concepts and its visibility rules.

This paper contains a description of the problems involved in mapping concurrent software onto distributed hardware environments excluding the trivial possibility of purely manual distribution. The feasibility of several alternatives is examined. Solutions to the problems (in particular, a description of possible implementations) is beyond the scope of this paper.

2. The Hardware Environment

The main goals, which are to be achieved with distributed systems, are increases of reliability, modularity, expandability, and adaptability of computer systems. A secondary goal, which seems to be obvious, is an increase of computing power and speed. However, the necessary communication between various parts of the system also increases and tends to degrade the performance of the system. Therefore the goals can only be achieved by

> an efficient distribution of the hardware components with respect to the requirements for reliability, modularity, and expandability

and by

> an efficient distribution of the software components with respect to a minimization of the necessary communication.

The distribution of hardware components may impose restrictions on the distribution of software components and vice versa.

The distribution of hardware is well understood, but there is little understanding of efficient software distribution and no sound theory exists. Software systems are normally tailored to specific hardware structures. This approach neglects the fact that changes of hardware components happen quite often, which then result in software changes or even partial redesign of the software system. It is now commonly recognised that software components are the more expensive parts in computer systems. Software changes should therefore be kept to a minimum. A system which distributes a hardware independent application program onto a formally described distributed environment would be the best solution. To achieve this goal, it must be clear:

> what range of target environments is relevant (i.e. the term "distributed system" must be defined),
>
> what the essential elements of distributed systems are (with respect to the problem of distributing a software system), and
>
> how such a distributed hardware system can be described formally.

This paper is primarily concerned with distributed systems in real-time environments. Such systems should be able to react to external events rather quickly. Therefore the work load should be equally distributed over the whole system.

2.1 Range of Distributed Systems

A distributed system can roughly be described as a network whose nodes are processing units, memory units, peripheral units, etc., and whose edges are the necessary communication links between the various nodes. Not every node has to be linked to every other node in the network. However, computers, which are not classed as distributed systems, are composed from peripherals, i/o processors, memory units, etc. Such a definition is therefore not precise enough.

To distinguish distributed systems from other systems, the terms "multi-computer system" and "multi-processor system" used to be widely used to describe the

architecture of distributed environments. They are, however, not of any help, because distributed systems incorporate both kinds of architecture, and because even "normal" computer systems consist of more than one processor (or even computer) nowadays.

A better definition of the term "distributed system" is given in [6]:

> A distributed system consists of a collection of highly autonomous nodes, with each node containing a processor (perhaps processors), primary storage, (perhaps) secondary storage, and a means whereby a node may communicate with its environment (e.g. terminals).
>
> Nodes are connected together and communicate via a communication network which exhibits variable (and perhaps unreliable) delays in the transfer of information from node to node.

This definition is sufficiently general to include both, distributed systems which consist of an homogeneous collection of processor nodes, each operating under control of a replicated but common operating system, and a variety of networks which may contain passive nodes as well as active nodes.

Note that the definition

(a) does not allow the inclusion of common memories (accessible at instruction level by several node processors), and

(b) implies the use (as the various nodes are highly autonomous) of distributed (rather than centralized) mechanisms for synchronization and co-operation.

The latter two features distinguish a distributed system from the more classic multi-processor system. The term "multi-computer system" is also excluded, because it implies the possibility of "master computers" controlling "slave computers" and therefore parts of the system or even the whole system.

Additionally the definition does not characterize (and hence not restrict) the class of applications for which a distributed system might be employed.

In summary, viewing distributed systems as collections of nearly autonomous nodes generally implies that one node may request service or information from another node, but that no node may completely explicitly control the behaviour of another. This further implies that a node cannot (in normal conditions) work stand-alone, because it needs information from other nodes in the system to fulfill its tasks. A distributed system (seen as a "black box") behaves to its environment like a "normal" computer system.

2.2 Essential Hardware Elements

Whether a hardware element of a distributed system is essential or not, depends on the purpose of the distributed system and on the person who at the moment deals with the system. This paper is only concerned with the task of distributing a given software system onto a given distributed hardware system. The distribution which will minimize the load on the necessary communication links between distributed hardware components and will maximize the parallelism of processes (Activities in MASCOT; tasks in Ada) hosted by the nodes is to be done automatically. Restrictions might be imposed by peripherals which are

connected to particular (processing) units (nodes of the network) only. Therefore only those hardware elements of a system are essential, which affect the distribution process. Further restrictions might be imposed by the characteristics of underlying runtime environments and the implementation language (i.e. the use of global objects in an Ada program). Restrictions caused by software are dealt with in the remainder of this paper.

The definitions given in subchapter 2.1 mention two major components of distributed systems: nodes, which host the computing power of the system, and links between the nodes, which establish a communication network within the system. It must therefore be assumed that these elements are essential. They would be sufficient for a distribution process, if every node had an unlimited size of main memory to store the programs running on it, had unrestricted access to all devices which are used to communicate with the system's environment, and were fast enough to process its data in realtime. In practice, nodes will have a limited computing speed, a limited size of main memory and will only have unrestricted access to those devices directly attached to them. Nevertheless, the distribution process is mainly guided by the number of nodes and of communication links. Decisions must take account of various restrictions. Restrictions therefore are attributes of nodes and communication links. An implementation of the distribution process may define any number of attributes with respect to special requirements.

In the remainder of this paper it is assumed that nodes consist of one processor only. The assumption is valid, because this paper only tries to solve the problems which result from synchronization of different nodes. The synchronization problems within a particular node must be resolved by the respective operating system. Routines of this operating system which handle the communication between nodes are common to all processors in the node. It is also assumed that all communication links between nodes work bidirectional.

As far as the problem of distributing a given software system onto a formally described distributed hardware environment is concerned there are two essential components of the hardware system: nodes and communication links between nodes. Both components have attributes which describe the limited nature of the nodes and the communication links. The use made of the information depends on the implementation of the distribution process.

2.3 Description Method

To be able to distribute a given software system onto a particular hardware system automatically, the hardware system should be described formally. The description is one input to the distribution process.

Every essential hardware element of the distributed system is identified by its type. The distributed system is represented by a graph. The nodes of the graph are the nodes of the distributed system. The edges of the graph are the bidirectional communication links. Nodes and edges have attributes which carry further information.

Possible node attributes are:

- number and type of processors
- size of main memory

- performance (i.e. processing speed)

- special memory locations (interrupt addresses, etc.)

- attached devices

Possible attributes of the communication links are:

- transfer rate

- type of link (serial, parallel)

The various attributes subdivide into mandatory and optional attributes. An implementation will define the appropriate attributes in both lists. The number and type of devices attached to a particular node seem to be mandatory attributes. However, a particular system may allow every node to have access to every device of the system by using a special communication network.

The following data types can be used to describe the various hardware components of the distributed system. The number, type, and value of attributes are implementation dependent. The type and value of "node.identification" and "interconnection.identification" also are implementation dependent and should state the unique system name of the particular node or of the particular communication link respectively.

```
TYPE attribute IS   -- implementation_defined;
TYPE attribute_list_item;
TYPE access_attribute_list IS ACCESS attribute_list_item;
TYPE attribute_list_item IS
   RECORD
      item : attribute;
      next : access_attribute_list;
   END RECORD;


TYPE node;
TYPE access_node IS ACCESS node;


TYPE interconnection;
TYPE access_interconnection IS ACCESS interconnection;


TYPE interconnection_list_item;
TYPE access_interconnection_list IS ACCESS interconnection_list_item;
TYPE interconnection_list_item IS
   RECORD
      item : access_interconnection;
      next : access_interconnection_list;
   END RECORD;
```

```
TYPE node IS
   RECORD
      identification           :   -- implementation_defined;
      list_of_attributes       : access_attribute_list;
      list_of_interconnections : access_interconnection_list;
   END RECORD;


TYPE interconnection IS
   RECORD
      identification     :   -- implementation_defined;
      list_of_attributes : access_attribute_list;
      node_1             : access_node;
      node_2             : access_node;
   END RECORD;
```

The hardware of a particular distributed system is represented by a list of nodes and a list of edges ("interconnections"). The actual structure of the lists is left to the particular implementation.

3. The Distribution of Application Programs

It is obvious that a distribution process needs a proper representation of the software to be distributed as well as the formal description of the hardware environment. The distribution process decides on the basis of both inputs how to split the application program into proper parts. However, before the distribution process itself can be defined, the distributable parts of an application program must be determined. Application programs developed according to MASCOT are discussed first. The problems involved in splitting an Ada program are dealt with afterwards.

3.1 Distributable Parts of MASCOT-like Programs

The MASCOT philosophy clearly defines the parts that an application program is built from: Activities and Inter-Communication Data Areas (IDAs). In theory Activities and IDAs are totally independent from other Activities and IDAs. The implementation of Activities or IDAs can be replaced by another implementation without affecting the construction of the application program, as long as the respective interfaces are left unchanged. To ease the design of larger application systems, logically closely related Activities and IDAs can be conceptually grouped together to form Subsystems. Therefore Subsystems too are to be considered as distributable parts. Moreover, in contrast to Activities, which are the subjects being scheduled by the runtime executive of a MASCOT Machine, Subsystems can be started, halted, resumed, and terminated by respective user actions during runtime (refer [2]).

Before concluding, how the parts of MASCOT-like application systems should be distributed, the characteristics of MASCOT units are discussed and their impacts on the distribution process outlined. Advantages and disadvantages are stated. Additionally, the impact of library units (i.e. pieces of software which are used by more than one Activity, etc.) is investigated.

MASCOT Device Handlers are not considered to be distributable parts, because they are closely interrelated with specific peripheral devices attached to particular nodes. They must therefore be loaded into the respective nodes, and that may constrain the distribution of other parts.

3.1.1 Inter-Communication Data Areas

Inter-Communication Data Areas serve as communication interfaces between Activities. Because Activities execute in parallel, their access to IDAs must be synchronized to ensure the integrity and consistency of the application system. MASCOT defines the synchronization mechanisms as part of the IDAs. The mechanisms are based on an asynchronous model, which is very helpful as far as distributed systems are concerned. IDAs are passive elements. Their execution is only triggered by Activities which want to use them. IDAs consist of a data area and access procedures to this data area. The access procedures implement mutual exclusion of those Activities which compete for access to the data area. The IDAs must be treated very carefully, because they influence the distributed synchronization mechanisms. Their location in the network must be chosen to keep the network-wide needs for synchronization to an absolute minimum.

ecause IDAs are the passive elements, their distribution should be considered fter the distribution of the active elements. Their location in the nvironment should be determined with respect to efficiency. Before the final ocation of an IDA is determined, the following options should be resolved:

the IDA can be loaded in the same node as the Activity which calls the IDA most often;

the IDA can be loaded in the same node as the Activity which is the consumer (or producer) of messages sent through it;

the IDA can be loaded in the same node as the Activity which must be served without any delay if it accesses the IDA;

the IDA can be loaded in the same node as the majority of those Activities using it;

the IDA can be loaded in such a way that the overall response time in accessing it is minimized throughout the network.

:t may be necessary that IDAs must be redistributed following results of initial :est runs, because the system has performed badly. This is possible, because distribution process makes its decisions on static assumptions. Dynamic .on.. derations can be achieved by simulating the behaviour of the application system with respect to the workload of the IDAs, taking the result of the distribution process as a first input. An investigation, to decide whether simulation or test runs with the newly built system is the better approach, is beyond the scope of this paper, because both a working distribution system and a proper simulation system are needed to provide necessary data on which to base a conclusion.

3.1.2 Activities

Activities are the active elements in MASCOT-like application systems. An aim of a distribution process should therefore be to maximize parallelism in the execution of the Activities. That means that every node of the distributed environment should always have an Activity in the running state. Ideally the number of idle nodes should be zero.

A distributed system is represented by a graph (refer 2.). Its active elements (e.g. processors) are the nodes of the graph, its passive elements (e.g. links between processors) the edges. A MASCOT-like application system is described by an ACP Diagram which is a graph with Activities and IDAs as nodes and links between Activities and IDAs as its edges. It is possible to transform an ACP Diagram into another graph showing Activities as its nodes and the connections between Activities via IDAs as its edges. IDAs themselves do not appear in this graph. The task of the distribution process is to reduce this graph to a graph which is equivalent to the graph representing the distributed system. The reduction, however, must maintain a maximum of parallelism between Activities and must ensure a minimum of communication between the nodes of the distributed system. Therefore it must be aware of the IDAs through which the various Activities communicate and it must distribute the IDAs accordingly.

An Activity is implemented by a reasonably small piece of code. The resulting code quantity can therefore be distributed in such a manner, that the amount of unused space of the main memories of the various nodes is minimized.

lication program is distributed in such a way that only one process of a node
municates with only one process of another node. This approach, however, may
n that highly competitive processes reside in the same node.

ther approach is feasible. Moreover, the locations of processes may be fixed
their needs to communicate with the system's environment through devices
ached to particular nodes. Therefore the following strategy is recommended.

(1) Recognise the processes whose locations are fixed by their needs to communicate with devices attached to particular nodes and distribute them accordingly.

(2) Determine the communication requirements of these processes with other processes and distribute the latter to achieve a minimum of load of the communication network. The distribution may be constrained by the use of global data, subprograms, etc.

result of the distribution process should be treated as a recommendation
ch must be approved by the development engineer. The final distribution
uld be made according to results of test runs, because inefficiencies caused
the communication requirements cannot be found statically.

3.2 MASCOT-like Programs

graph-like representation of MASCOT-like programs is maintained by the MASCOT
nstruction Data Base. The representation is equivalent to the ACP Diagram of
program on the uppermost level. Additionally, it shows from what templates
ivities and IDAs are derived.

is recommended to take Activities as the basis for the distribution of
SCOT-like application programs. The disadvantage which occurs in addressing
systems is minor, because the distribution process will put closely related
ivities (normally those of a Subsystem) together in the same node. The
vantage gained is a better and more suitable distribution of the whole
gram.

As should be loaded in the same node as those Activities which access them
st to minimize the load of the communication network. However, a particular
lementation may choose another approach or leave it flexible so that the
stem's constructor can choose the strategy interactively.

distribution process should be supplied manually (probably interactively
er it has made initial decisions) with some guidelines for the distribution
special objects.

the MASCOT Machine an application program runs on is an Evolutionary Machine,
omatic distribution of the application program is not feasible, since the
gram may change online. The location of the various program parts must
refore be specified by the system's constructor.

3.2.5 Input-Output

To allow a program to communicate with its environment, Ada provides a set of input/output packages (refer [1]). These packages must be loaded in those nodes, to which the particular external devices are connected. The code of the input/output packages can be duplicated as often as there are different devices attached to the system. The distribution of the program units referring to particular input/output packages is constrained by the location of the packages.

If special input/output routines are used in an application system, these routines should be embedded in special packages for each device. Such an approach eases the distribution.

3.3 The Distribution Process

Basically the task of the distribution process is to reduce a proper graph representation of an application program (the ACP Diagram in the case of a MASCOT-like program; the scope tree with its use and call relationships in the case of an Ada program) to the graph representation of the distributed hardware environment. The methods of graph reductions are many and are not presented in this paper.

An actual implementation of the distribution process must be based on adjustable distribution requirements and strategies. The distribution of an application program is further constrained by its input/output needs.

The results of the distribution process are inputs for the code generation phase of compilers, for linkage editors, and perhaps for loaders of the distributed environment.

3.3.1 Requirements and Strategies

The distribution process is mainly confronted with two conflicting requirements:

(1) minimization of the load of the communication network, and

(2) maximization of the parallelism of processes (Activities in MASCOT; tasks in Ada).

Both requirements are essentially to achieve a behaviour of the distributed system which suits a realtime environment where a low response time to external (and also internal) events is disastrous. Decreasing the load on the communication network and increasing on parallelism of processes improves response time.

A maximum of parallelism can be achieved by allocating each process to a node of the distributed environment. This is, however, impossible, especially if an Ada program is considered where the number of actually created tasks cannot be determined statically. Additionally, a huge number of nodes would be required, of which at a given time most would be idle.

A minimum of load of the communication network can be achieved, if the

.2.4 Dynamic Creation of Task Objects

'ask objects are special objects in Ada: a fact which complicates the istribution process. Tasks are not the units of which an Ada program is onstructed. The structural components of an Ada program are subprograms and ackages (library units in a more common sense). Tasks are declarative items ithin library units. They can be declared wherever a declarative part is llowed. Nevertheless they are program units and define a sequence of actions. 'ask objects come into existence when the respective program unit or block tatement starts to execute its sequence of statements. A task object ceases to xist before the respective program unit or block statement, in which the type f the task object is declared, terminates the execution of its sequence of tatements. Task objects can even be created dynamically by the elaboration of llocators, if they are derived from task access types. Ada does not restrict he creation of task objects in any way. Therefore it is not predictable from a tatic point of view, how many task objects will exist during the runtime of an da program. This kind of flexibility is not helpful to the distribution of Ada rograms. However, to allow for the full power of the Ada language, estrictions should not be put on programmers by a distribution process.

he declaration of a task type consists of two parts: the declaration of its nterfaces (entries), which define means by which a derived task object can ommunicate with other task objects in a controlled manner, and the declaration f its body, which defines the sequence of actions (code) a derived task object ill perform. The body of a task type is reentrant, a fact that can be utilized y a distribution process. It allows that the code of a task type can be uplicated within the distributed environment. Problems arise, if the task body ses global objects. The use of global subprograms is not critical. These subprograms are unable to use objects of the task body. Therefore control can e transferred to them, even if they are located in another node of the nvironment. On the other hand, if a referenced subprogram only uses local eclarative items, its code can be duplicated and loaded in the node the task bject runs on.

ask objects can be declared within a task body. A distribution process should reat them as being declared on the same level as the parent task. Only their life time is constrained by the parent task.

f tasks are components of record objects or array objects, they start to live immediately after the elaboration of the respective record object or array bject.

distribution process can only work on static terms despite the fact, that the ives of Ada tasks rely on dynamic terms. Therefore it can only take the number f task types, the number of task objects of anonymous task types, and the umber and occurrences of task object declarations and of allocators as base for distribution. If several tasks are created consecutively by the same eclaration or allocator due to runtime behaviour, the resulting task objects should be located in the same node of the distributed environment. In the case f task access types, the task objects must be located in the node in which the rogram unit resides in which the respective type has been declared.

he distribution of a task is restricted, if one of its entries is connected to hardware interrupt.

either transfers control to a subprogram by issuing an appropriate subprogram call or wants to rendezvous with a task object by issuing an appropriate entry call. In either case a program unit only uses an interface of another unit. Call relationships therefore denote direct communication paths between different parts of application programs. If the called program unit is declared within a package declaration, a use relationship is automatically established between the calling unit and the respective package declaration.

The various use relationships form a dependency graph of an application system. It is obvious that the graph consists of several unconnected subgraphs, called relationship graphs. These relationship graphs denote disjoint scope areas of the application program, i.e. parts of the Ada program which are totally independent from each other, apart from call relationships or from the initiation of task objects which actually establish these scope areas.

The disjoint relationship graphs determine the distributable parts of an application program. The distribution may only be constrained by call relationships which represent subprogram calls, if the addressed subprograms use global items (refer 3.2.1.4).

3.2.2 Subunits

The declaration of subprograms, packages, and tasks can specify that the associated bodies are to be compiled separatly. The program units are then called subunits. The program units containing the declarations are called parent units. A subunit is effectively part of its parent unit and must mention its parent unit. The language facility only eases compilation. The distribution process must consider a parent unit together with its subunits.

Subunits, however, may import library units in a context clause. These library units have to be considered additionally.

3.2.3 Library Units

Subprograms, packages, and generic declarations and their bodies as well as generic instantiations can be compiled on their own. These units (except their bodies) are called library units in Ada. Bodies can only be compiled after successfull compilation of the respective subprogram specification, generic declaration or package declaration.

Library units can import other library units (i.e. make the respective interface visible). In this case library units depend on other library units. Because of recompilation restrictions (refer [1]) an Ada compiler builds a dependency graph which represents the visibility connections between library units.

The distribution process builds a scope tree for every library unit. The dependency graph of the library units is used to connect the various scope trees and to form the relationships between the various nodes of the scope tree.

The resulting disjoint relationship graphs determine the distributable parts of the overall program.

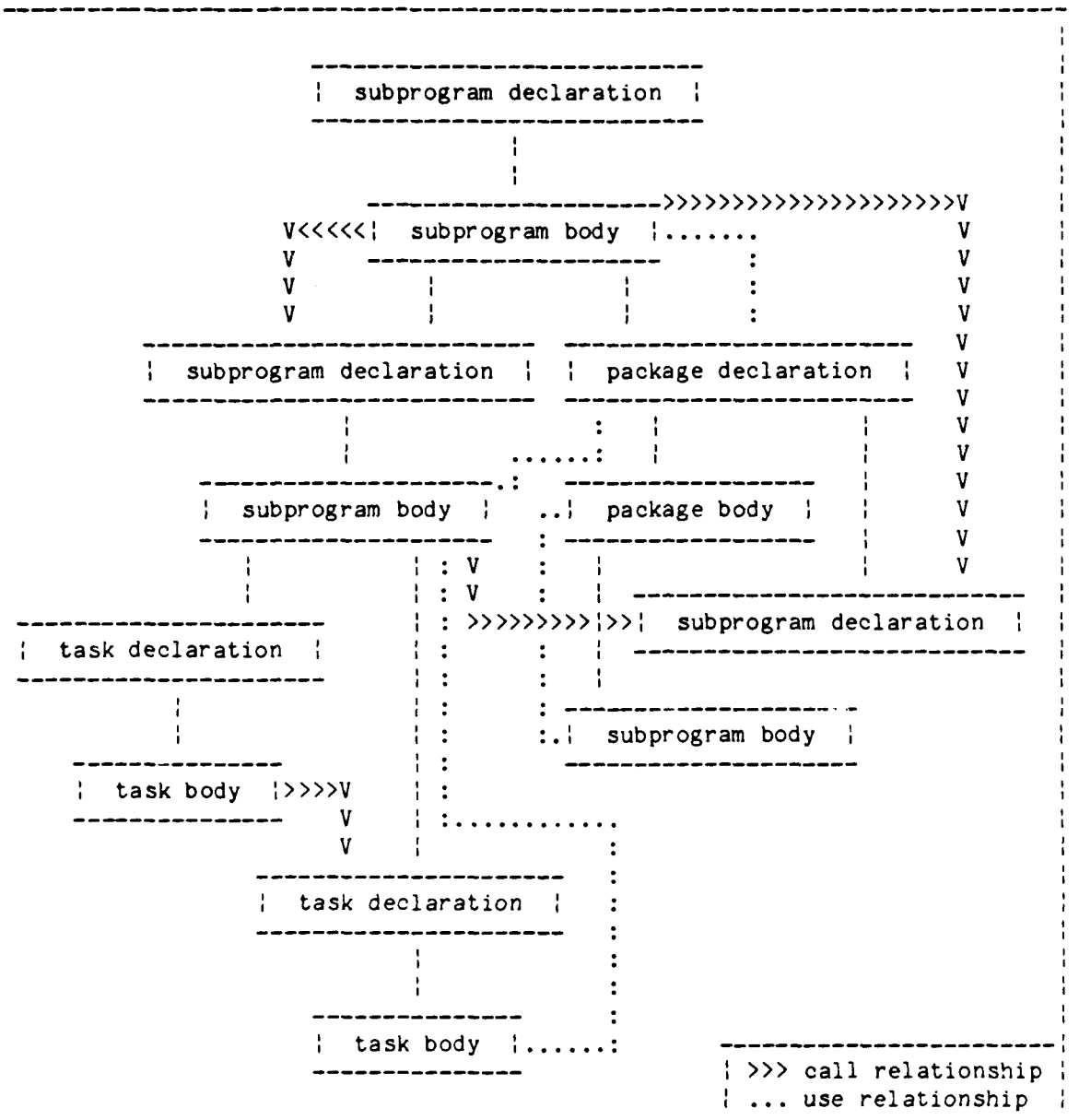


Figure 3.2-6 : Scope Tree and Relationship Graphs

Nodes of the scope tree may contain two types of relationships to other nodes:

(1) use relationship

A use relationship is established between two program units, if one of the units uses objects or types declared in the other unit. Use relationships constrain the distributability of application programs. They also denote possible indirect communication paths between parts of application programs.

(2) call relationship

A call relationship denotes, whether a program unit or block statement

declared in its enclosing scope. If it does so, these objects may be shared by several tasks and are not protected from competing accesses, unless the access to them is put in critical regions.

A task starts its execution at the point where its immediately enclosing program unit commences to execute its sequence of statements or, in the case of a task access object, after evaluation of the respective allocator. A task object will end its execution and will be destroyed afterwards when the scope of the program unit is left in which its type is declared. However, the enclosing program unit must wait until the task has finished to execute its sequence of statements.

Task objects form the basis for the distribution process, because they are the only active elements in an Ada program. If an Ada program does not contain tasks, it is not distributable. The procedure which denotes the main program is treated like a task object. Because the existence of tasks depends on the execution of the particular Ada program, the distribution process has to evaluate the whole program to determine the number of task type and task object declarations. After this computation the splitting of the Ada program is done according to its scope hierarchy.

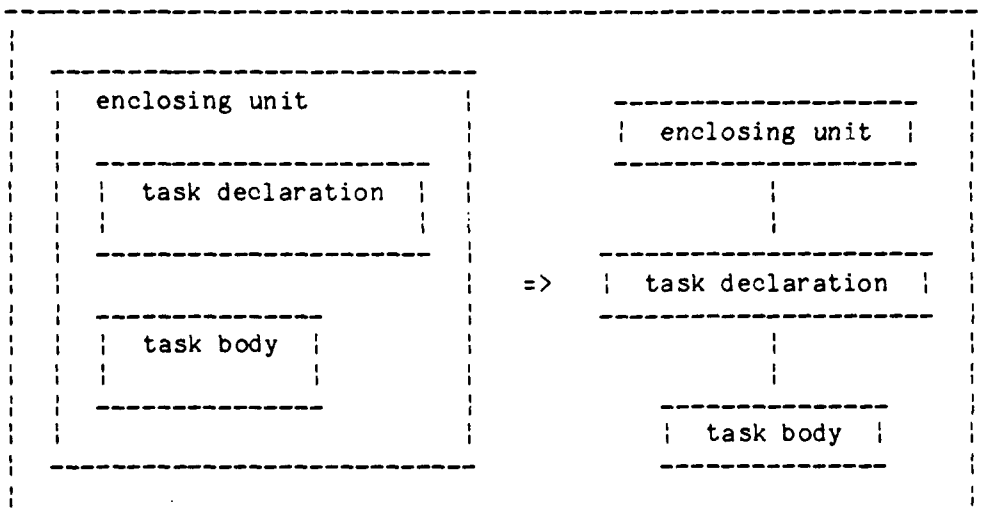


Figure 3.2-5 : Transformation Rule for Tasks

3.2.1.9 Scope Hierarchy and Relations between Scopes

An Ada program establishes a hierarchy of scopes, which can be represented by a tree structure (scope tree). Every node of the tree represents either the interface (VOID in case of a block statement) or the scope of a subprogram, package, task, or block statement (refer to the transformation rules shown by Figures 3.2-1 to 3.2-5). An example of a scope tree is shown by Figure 3.2-6.

and because a package is only elaborated once in an application program due to language rules, the code of a package body can only exist once in a distributed environment.

The transformation rule is shown by Figure 3.2-4. All possible transformations are demonstrated.

3.2.1.6 Generic Declarations

A generic declaration effectively specifies a template either for a set of subprograms or for a set of packages.

A generic unit is written as a subprogram or package but with the specification prefixed by a generic formal part which may declare generic formal parameters. A generic formal parameter is either a type, a subprogram, or an object.

Generic units are not considered by the distribution process.

3.2.1.7 Generic Instantiations

A subprogram or package created using a template is called an instance of the generic unit. A generic instantiation is the kind of declaration that creates an instance.

Instances of generic units are treated as packages or subprograms respectively by the distribution process.

3.2.1.8 Tasks

A task operates in parallel with other parts of the program. It is written as a task specification (specifying the name of the task and the names and formal parameters of its entries) and a task body which defines its execution.

A task type is a type that permits the subsequent declaration of any number of similar tasks of the type. A value of a task type (task object) is said to designate a task. A task object which is not derived from a task type is said to be of an anonymous type specified at the point of the task declaration. Access types of task types are also possible. Tasks can be part of compound objects (record objects, array objects).

Task entries define the interfaces which allow the passing of messages between tasks. They are points of synchronization of task objects. The message handling is based on a synchronous model. The interested reader is referred to [1] for more information.

A task body may contain a declarative part. All items declared therein are hidden from outside the task body. Therefore the declarative part establishes a new level in the scope hierarchy of an Ada program. This scope is a leaf of the node representing the scope of the declarative part where the respective task type is declared in (refer Figure 3.2-5). A task body may refer to items

distributability of dependent program units is not affected.

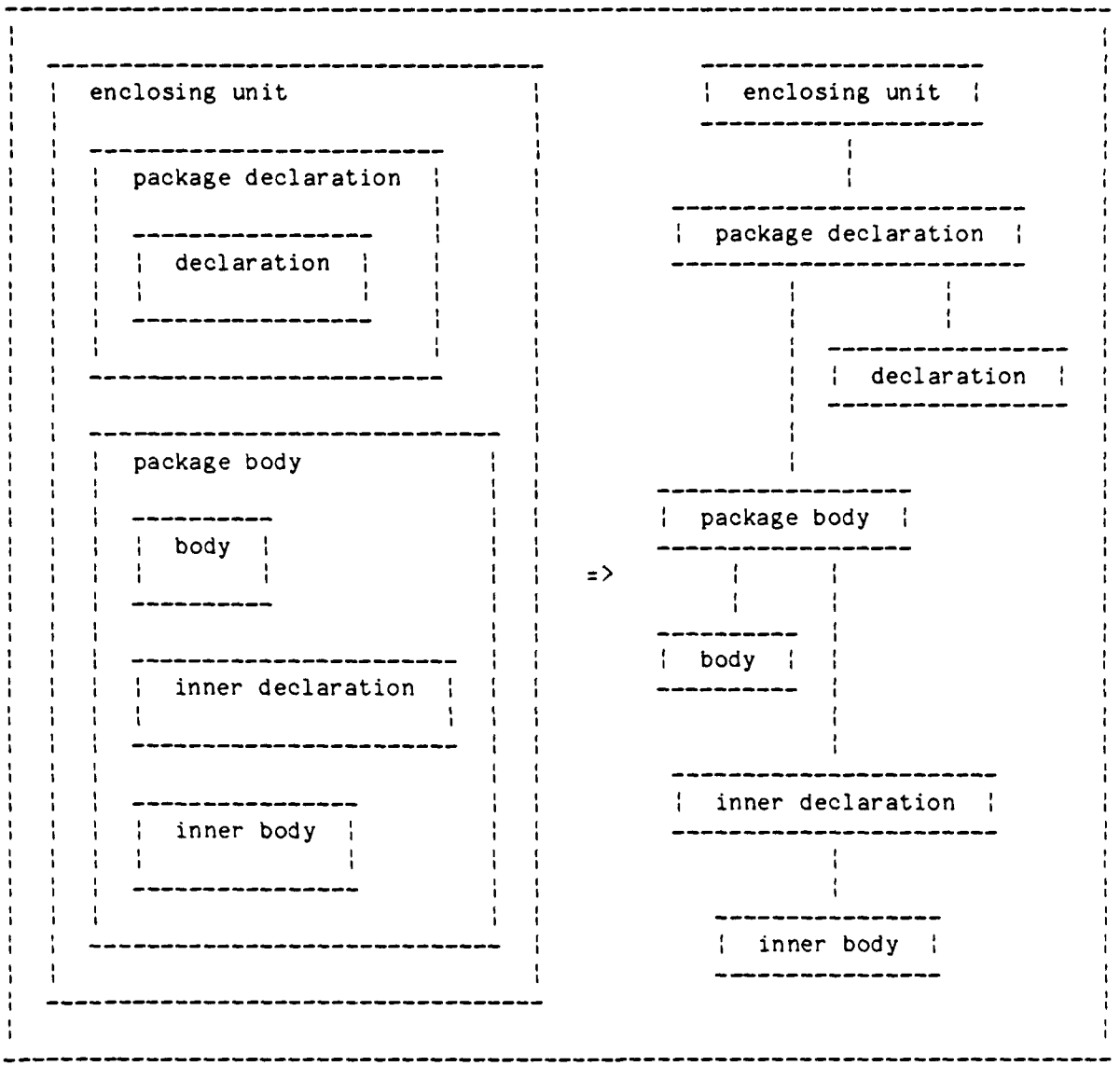


Figure 3.2-4 : Transformation Rule for Packages

A package body (which is optional in the case, where the declaration does not contain subprogram, task, and package declarations) consists of a declarative part and an optional sequence of statements. The sequence of statements is executed as part of the elaboration of the package. Its main purpose is to initialise objects declared in the package. The package body is elaborated together with the package declaration. Items, which are declared in the package body additionally to those declared in the package declaration, are hidden from outside the package. They are part of the implementation of the visible items. They may form a kind of memory of the package, because their values can only be altered by using the facilities provided by the package. A package body therefore establishes a new level in the scope hierarchy. The level is a leaf of the node representing the enclosing scope. Because of the memory facility

and the overall subprogram is still independent of its environment. Otherwise, copying the body is not possible, and that leads to problems for the distribution process.

The transformation rule for subprograms is shown by Figure 3.2-3.

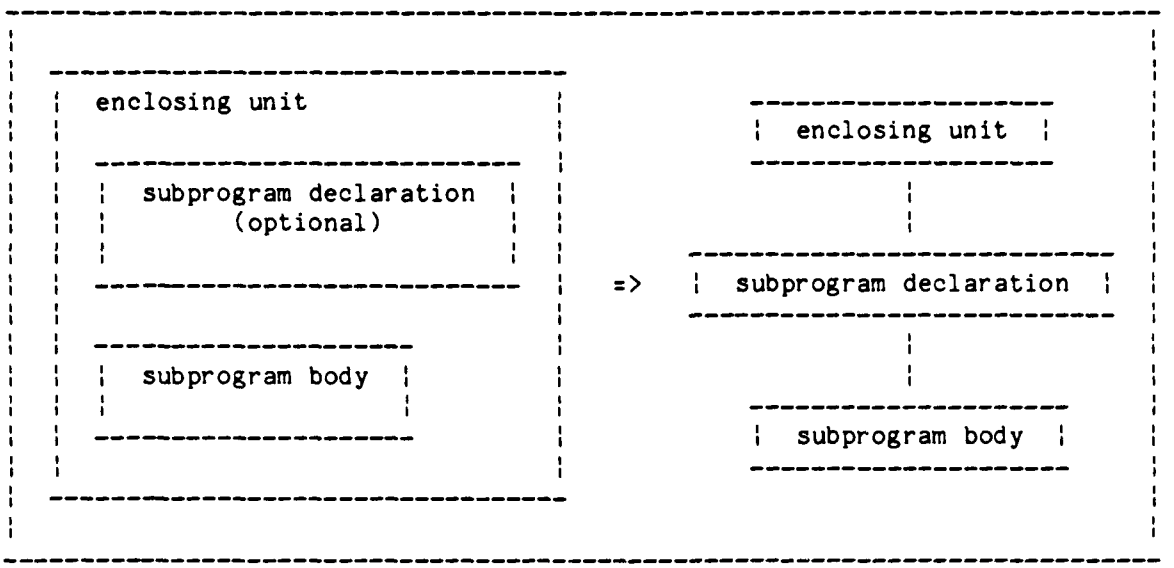


Figure 3.2-3 : Transformation Rule for a Subprogram

3.2.1.5 Packages

A package specifies a group of logically related items, such as types, objects of those types, and subprograms with parameters of those types. It may also specify tasks and other packages. It is written as a package declaration and a package body. The package declaration has a visible part, containing the declarations of all items that can be explicitly used outside the package. It may also have a private part containing structural details that complete the specification of visible items, but which are irrelevant to the user of the package. The package body contains implementations of the subprograms, tasks, and other packages that have been specified in the package declaration. The package body may declare items additionally to those specified in the package declaration.

For the purpose of this paper a difference between visible and private part of a package declaration is not necessary. The package declaration is elaborated once. After the occurrance of the package declaration all items declared in it can be used within the scope of the declarative part that the package declaration appears in. A package declaration therefore does not introduce a new level in the scope hierarchy. It may, however, introduce global items. If a package declaration contains global items, it restricts the distributability of other program units which depend on the package declaration (i.e. which use items declared in the package declaration). If the package declaration only contains type declarations (however, not access or task type declarations), the

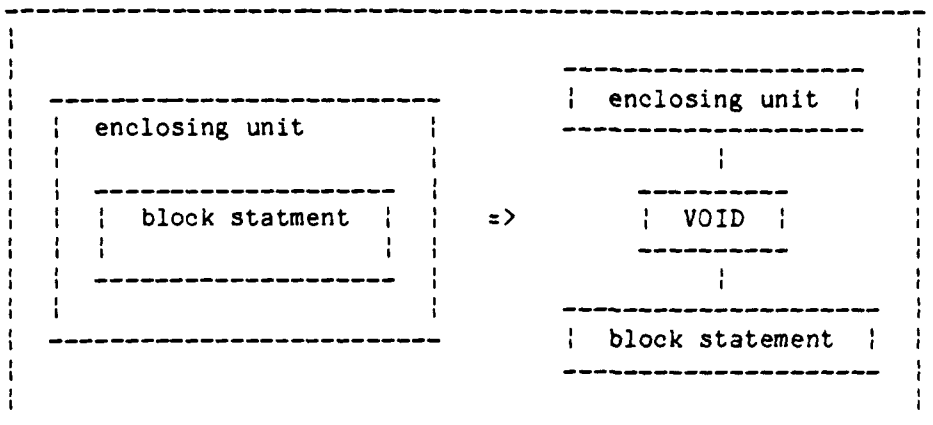


Figure 3.2-2 : Transformation of a Block Statement

Block statements themselves are not considered to be distributable parts of Ada programs. Problems, however, arise for the distribution process, if the declarative part of a block statement contains the declaration of task objects.

3.2.1.4 Subprograms

In Ada, a subprogram is written as a subprogram declaration (specifying its name, formal parameters, and, in the case of a function, the type of the returned value) and a subprogram body (specifying the sequence of actions). The subprogram declaration is optional.

The subprogram body may contain a declarative part. Items declared in this declarative part are not visible outside the subprogram. On the other hand, items, declared prior to the subprogram body in the same declarative part or in enclosing scopes, are visible inside the subprogram. The declaration of a subprogram body therefore introduces a new level in the scope hierarchy of an Ada program. If two subprograms are specified in the same declarative part, the scopes introduced by them are disjoint. If the scope hierarchy is represented by a tree-like structure, the scopes of the two subprograms are leaves of the node built by the declarative part the two subprograms are specified in.

In Ada, subprograms are reentrant. This can be achieved because the declarative part of a subprogram is elaborated at the point of its call. Therefore the scope of a subprogram is not effective until the subprogram is called. If a subprogram uses only items local to it, its execution is totally independent of its environment apart from actual subprogram parameters. Therefore the existence of several instances of such a subprogram in different nodes of a distributed system would not change the behaviour of an application system. This fact can be utilized by the distribution process.

However, if a subprogram refers to global items, it may not be independent of its environment. The independence of the subprogram depends on the type of the global items used. If the global items are other subprograms which themselves do not use global items, the bodies of these subprograms can also be duplicated

program, perhaps to point out weaknesses in the program design.

3.2.1.2 Type Declarations

Type declarations define a set of values and a set of operations applicable to those values. Objects (e.g. program variables) or other types can be derived. from a type. Types do not introduce problems for the distribution of a program, because their only purpose is to define the characteristics of derivable objects. Ada, however, embodies two types which must be treated carefully: task types and access types. Task types are dealt with later in this chapter.

Access types enable the definition of reference objects. A value of an access type (an access value) is either a null value (if there is no object which can be referred to) or a value that designates an object created by an allocator. The designated object can only be read and updated via the access value contained in an access object. The definition of an access type specifies the type of the objects designated by values (objects) of the access type. An access type declaration establishes a memory area in which all objects created by evaluation of allocators for an access type are stored. Designated objects may be created dynamically during runtime. The reserved memory area for these objects is associated with the program unit in which the access type is declared. Designated objects stored in the area exist, until the program unit is left. A designated object may therefore live longer than an object which holds a respective access value. Designated objects should be treated as global objects by the distribution process.

3.2.1.3 Block Statements

A block statement is a compound statement that may contain a sequence of statements. It may also contain a declarative part. The effects of the declarative part are local to the block statement. The block statement can use items declared in enclosing scopes. A block statement introduces a new level in the scope hierarchy of an Ada program (refer Figure 3.2-2). It defines no interface.

declarative item. This association is in effect within a region of the program text, called the scope of the declaration. Within the scope of a declaration, there are places where it is possible to use the identifier to refer to the associated declared item. At this places the declaration is said to be visible. For a precise definition of the scope of the various declarative items and the associated visibility rules the interested reader is referred to [1].

An Ada program is a hierarchy of declarative regions (refer Figure 3.2-1). In general an inner scope may use items declared in its enclosing scopes. The immediately enclosing scope can only use the interface of the inner unit, defined by the specification of the inner unit, if the service of the inner unit is required. The enclosing unit cannot use items of the body of the inner unit.

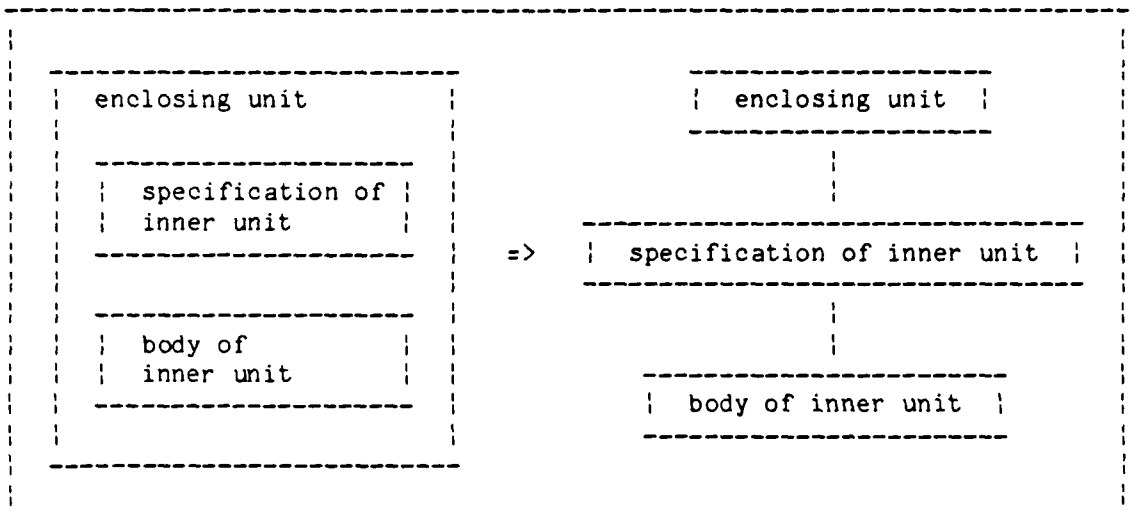


Figure 3.2-1 : General Transformation Rule

3.2.1.1 Object Declarations

An object declaration defines an item of a program which contains a value. Objects are derived from types and can only contain values of that type.

If objects, which are declared in enclosing scopes, are used in an inner scope, these objects are said to be global to the inner scope. They introduce difficulties, if a program is to be distributed. They may be referred to by program units which are located in different nodes of the distributed system. In this case the distributed synchronization mechanism must ensure their integrity. Such objects restrict the distribution of associated program units.

If global objects are used by different task objects, the language does not define means by which their integrity and the consistency of the whole program is ensured, unless the global objects are declared to be shared (achieved by use of the pragma SHARED) or unless their use is embedded in critical regions which implement mutual exclusion of the respective task objects. Nevertheless, a distribution process should determine the global objects of an application

3.1.4 Library Routines

Application systems frequently utilize a number of common routines. Such routines are called from various locations of an application program. If the parts of an application program are spread over a distributed hardware environment, it is very likely that a particular library routine is located in a different node from the caller and that may cause a lot of inter-node communication and decreasing the system's performance significantly.

Library units can be separated into two different classes: library units having some sort of memory, and library units not having a memory. The latter class does not impose any problems, because the library units behave in the same manner in subsequent calls. Therefore they can be loaded by an application program more than once. The only penalty is a higher consumption of space in the main memory. In a distributed environment, however, they have the advantage that they can be loaded in those nodes where their service is required.

Library routines with memory behave differently in subsequent calls. They cannot be loaded by an application program more than once because the behaviour of the application program would change unpredictably. They do not cause any problems for application programs developed according to the MASCOT philosophy. Such library routines are hidden means of communication. The MASCOT rules, however, state that all communication has to be performed by using IDAs. Therefore such library routines are illegal in MASCOT and need not be considered by a process which distributes MASCOT-like application systems.

3.2 Distributable Parts of Ada Programs

An Ada main program is a sequential program which may consist of special objects, called task objects, to allow for parallel computation. Ada programs which do not contain task objects are not considered in the remainder of this paper. An Ada main program may additionally use library units made visible by its context clause. These library units may or may not contain task objects.

An Ada main program can only be distributed after static determination of all task objects it contains. Task objects, however, may be created dynamically. In this case the distribution process can only locate, where in the program such a task object is initiated. It cannot statically determine the number of task objects, which will eventually be created during runtime. The main program is considered to be a subprogram called by a task object of the underlying operating system.

3.2.1 Visibility and Scope of Declarations

Ada defines various items which can be declared in declarative parts and are of interest to a distribution process: types, objects, subprograms, tasks, packages, generic declarations, and generic instantiations. Subprograms, tasks, packages, and generic declarations may contain declarative parts themselves. An Ada block statement, which can occur everywhere where a statement is allowed in an Ada program, may also contain a declarative part.

A declaration associates an identifier (or some other notation) with a

MASCOT Device Handlers are not considered to be distributable (refer 3.1). To be able to communicate with the system's environment, Activities must eventually call particular Device Handlers. Such Activities should reside in the same node as the Device Handler for efficiency reasons. Therefore only a subset of the Activities is distributable.

The distribution of Activities is therefore constrained by four conflicting requirements:

(1) The location of Activities which depend on specific Device Handlers is fixed.

(2) The load of the communication network and the need for network-wide synchronization should be minimized.

(3) A maximum of parallelism between Activities should be achieved.

(4) The amount of unused main memory of nodes should be minimized.

These conflicts may be resolved by taking requirements (1) and (4) as initial considerations for the distribution, followed by requirement (3). Requirement (2) may be achieved by shuffling IDAs around accordingly.

It must be realised that only Subsystems can be started, halted, resumed, and terminated by a user of a MASCOT system during runtime. Therefore, if Activities are taken as basis for the distribution, a special table must be kept in the system. This table shows where the various Activities which belong to a particular Subsystem are located. Every user action that influences the behaviour of a Subsystem has to be performed network-wide which may result in a temporary decrease of the system's performance and may, in the worst case, result in several unwanted effects.

3.1.3 Subsystems

Despite the fact that Subsystems were introduced into MASCOT as a conceptual enhancement only, to ease the design of larger systems, they are the units which must be addressed, if anybody wants to influence an application system's behaviour online. The scheduler of a MASCOT Machine, however, only knows Activities as schedulable units. Nevertheless, it is worth considering whether Subsystems can be taken as a sound basis for the distribution of an application system.

Obviously the number of distributable items is reduced significantly. It could even be that a distribution by hand is done more quickly than by an appropriate program. If anybody wants to influence the execution of a Subsystem, only one node of the distributed system is affected. The communication in the distributed system is only influenced by the number and location of those IDAs which form interfaces between Subsystems. Therefore the possible number of conflicts within the communication network is reduced.

However, the application system might not be distributed as efficiently as it could be. Highly competitive Activities might be loaded in the same node, because they are part of the same Subsystem. Therefore a high degree of parallel computation is not achieved, and one of the major goals of distributed systems is missed.

3.3.3 Ada Programs

According to the Ada Language Reference Manual [1] an Ada program is represented by a procedure (called main procedure) which may use library units. The main procedure is treated as a task by the underlying runtime environment. During its execution the main procedure may initiate a statically undeterminable number of tasks. It is possible that tasks declared in the text of an Ada program are never initialized because the initialization depends on results computed during the execution of the Ada program. The distribution process must be aware of that fact and prevent a node from only hosting tasks which will never be used. The node hosting the main procedure works as a start-up node for the distributed system. The initialization of tasks can be treated as remote procedure calls where the calling node is allowed to continue execution but cannot terminate before the remote node will have terminated.

The distribution of an Ada program is achieved in several steps. The distribution process itself needs a proper representation of the distributed system (refer 2.) and a proper representation of the Ada program (probably a tree-like representation as produced by an Ada compiler as intermediate form at the end of the semantic analysis). The distribution process generates information for the code generation phase of an Ada compiler, the linkage editor, and the loader.

The four steps of the distribution process are:

(1) Derive the scope tree from the intermediate representation of the Ada program.

(2) Determine the use and call relationships between the various parts of the Ada program.

(3) Determine those subprograms which can be copied (i.e. the subprograms which do not use global objects) and enlarge the graph representation accordingly. Mark the disjoint relationship graphs (refer 3.2.1.9).

(4) Distribute the disjoint relationship graphs. Consider input/output constraints etc. of the various program units while distributing.

3.3.4 MASCOT/Ada Programs

If Ada is used for the implementation of application systems designed according to the MASCOT philosophy, one step of the distribution problem is already done: the design of the ACP Diagram. The ACP Diagram breaks an application system down into distributable parts. Nevertheless, a check should be made by a suitable program to discover hidden global objects that may stem from the use of library units with which the user is unfamiliar.

4. Communication

Each node of a distributed system must eventually communicate with other nodes in the system. The means of communication may be used to pass solicited or unsolicited information to other nodes, to request (initiate) action sequences in other nodes, or to synchronize progress, in some way, with one or more other nodes.

4.1 Communication Mechanisms

Programs which facilitate parallel computation make special demands on communications mainly for synchronization purposes. It was to accommodate these demands that monitor and other constructs were conceived for shared memory systems. For distributed systems (without shared memory) communication is made by sending messages between parts of the application system, which reside in different nodes.

Message sending may assume two distinct forms:

message passing, or

direct remote invocation.

Each form is associated with an accompanying construct with unique syntax and semantics. The construct is not visible at the user (application system) level as far as the problems discussed in this paper are concerned. As it is not clear, what construct is the most suitable in combination with MASCOT and/or Ada systems, representative constructs are briefly surveyed. The constructs may be seen to differ in terms of:

the effects on flow of control the construct imposes on the sending and receiving nodes, and/or

the nature of the synchronization imposed or assumed by invocation of the construct, and/or

the structure the construct imposes on the program which uses it (It may be at odds with the semantics of MASCOT or Ada).

4.1.1 Message Passing

Message passing schemes generally allow nodes to transmit requests for actions or data to other nodes. These requests are honoured by the receiving node according to the locus of control established by the runtime executive which controls that node. The receipt of a request does not forcibly alter the flow of control at the receiving node. Additionally the sending node may

immediately suspend processing and await a response, or

continue processing until such time as the response is required.

Before the receiving node generates a response, it may acknowledge the request.

Message passing constructs are probably the most versatile. They allow third party responses which the others do not (at least directly) and they explicitly consider timeouts and conditions related to damaged message delivery or failed delivery.

4.1.2 Remote Invocation

Remote invocation schemes allow processes residing in a node to instantiate (call) procedures defined in other nodes, with the attendant blockage of the requesting (instantiating) process until execution of that request is completed.

Remote procedure calls are not necessarily distinguishable from normal procedure calls (i.e. calls of procedures which reside in the same node as the caller) from the program text. Because remote procedure calls generally have the same syntax as normal procedure calls, their use is constrained by the following difficulties:

> They are asymmetric with respect to the delays imposed on sending and receiving nodes, i.e. the called procedure is not executed immediately after its call, and the execution of the caller is not resumed immediately after the termination of the remote procedure.
>
> They offer no provisions for timeout declaration or handling within the application program. The syntax and semantics of procedure calls do not allow the specification of such parameters and handlers.
>
> They force notification of improper transactions to be given the receiving node rather than the sending node, even if notifying the sending node may be more appropriate. The syntax and sematics of procedure calls do not allow the passing of error messages from the runtime environment to the application program.

Remote invocations are handled by the underlying runtime environment on the basis of message passing schemes. The runtime environment must handle errors which occur during the transmission of the necessary messages between sending and receiving node. If the programming language the application program is written in allows the specification of exception handlers within the text of the application program, information of failures can be passed to the application program and the errors can be handled there. Otherwise failure situations are not resolvable within an application program itself.

4.1.3 Paired Input/Output Statements

Communicating sequential processes may interact by means of paired input/output statements. A node executing a named input (output) is blocked until another node executes an output (input) with the same name. When a named pair of commands exist, information is exchanged, and both nodes are allowed to progress.

Paired input/output statements are equivalent to Ada's rendezvous, if the respective accept statement does not contain a sequence of statements.

4.2 Communication in MASCOT-like Systems

Active elements (Activities) of a MASCOT-like application system may only communicate via Inter-Communication Data Areas (IDAs). An IDA provides a set of procedures, which can be called by Activities, to gain access to its data area. These procedures implement critical regions to ensure mutual exclusion of competing Activities. It seems to be natural, that the procedures are invoked by remote procedure calls, if the IDA resides in a different node from the Activity which issues a call.

4.2.1 Invocation of Access Procedures

An Activity issuing a request for an IDA does not know where in the system the IDA resides. Therefore it must pass control to a special handler which keeps the locations (nodes) of the various IDAs in a special table. Every node has knowledge of only those IDAs which are referred to by Activities of the node. The execution of an Activity is suspended, after control has been passed to the handler. When an access procedure has finished its execution, the handler transfers control back to the Activity and the Activity resumes its execution.

The following algorithm defines the call of an access procedure in a distributed environment.

(1) An Activity calls an access procedure of an IDA.

(2) A proper handler accepts the call and the Activity is blocked.

(3) The handler determines the location of the IDA in the network.

(4) If the IDA resides in the same node, a handling instance of the Activity is generated, which calls the respective access procedure. When the access procedure has finished its execution, the handling instance is destroyed and control is passed back to the handler. The next action is defined by step (9).

(5) If the IDA resides in a remote node, the handler transfers the parameters of the access procedure to the communication network, additionally to the name of the IDA, the name of the access procedure and the name of the remote node.

(6) A handling instance of the calling Activity is generated in the remote node. The handling instance calls the respective access procedure.

(7) When the access procedure has finished its execution, its results are passed back to the handling instance.

(8) The handling instance passes the results back to the handler in the calling node via the communication network. After successful communication the handling instance in the remote node is destroyed.

(9) The handler activates the Activity.

Figure 4.2-1 shows the flow of control, if an access procedure of a remote IDA is called.

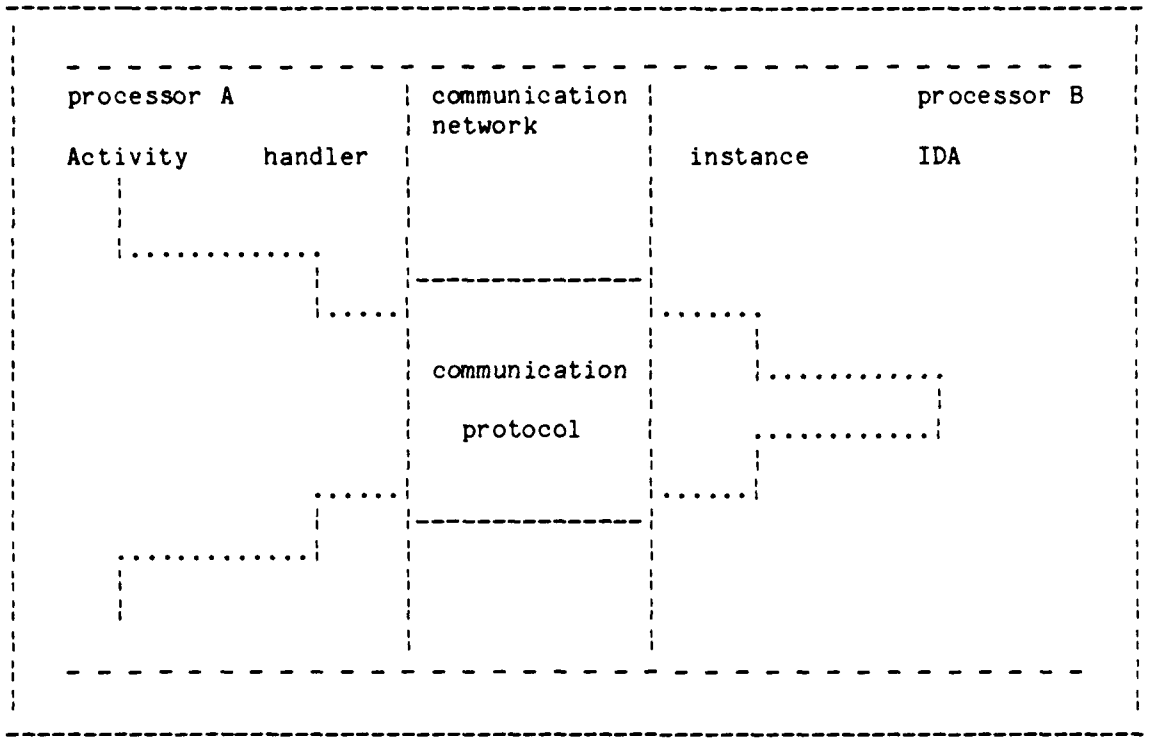


Figure 4.2-1: Remote Invocation of an Access Procedure

4.2.2 Problems of Remote Invocation of Access Procedures

The above algorithm begs the following questions:

> Are there any restrictions on the types of the parameters of an access procedure? How is the parameter passing done?
>
> How are failures of the communication network dealt with?
>
> Are the semantics of the MASCOT primitives still valid?

These problems are discussed in the following subchapters.

4.2.2.1 Parameters of Access Procedures

It is necessary to assume only three different types of parameters of access procedures: parameters, which are passed by copying their values, reference parameters (which include parameters for which only a reference to their memory location is passed), and parameters, which denote subprograms.

Copy parameters do not introduce any problems. Their values are transmitted

either with the remote invocation of the access procedure or when control is passed back to the calling node.

Reference parameters have as value the memory location of the objects they refer to. Therefore this reference is transmitted with the invocation. When they are used by the access procedure, the real value must be obtained or updated by the access procedure by using the communication network. By doing so, the execution of the access procedure is suspended. This may degrade the performance of the system. It should therefore be decided either not to use reference parameters or to copy the value of the object(s) they refer to assuming the location is within the calling node.

If names of subprograms are passed as parameters, the access procedure must issue a remote invocation of the respective subprogram, if it uses the parameter. This imposes a similar constraint as reference parameters.

It would therefore be wise to use copy parameters as parameters of access procedures only, otherwise realtime conditions may be violated.

4.2.2.2 Communication Problems

In distributed systems nodes cannot detect that other nodes do not work properly, unless the faulty nodes provide appropriate information themselves. Knowledge about faults can only be derived by transmitting requests to nodes assuming that the communication links still work. If such a request is not answered within a specified period of time it is fair to assume that the requested node does not work. In a MASCOT-like program the only possible occasion, at which such failure situations are determinable, is when an Activity calls a remote IDA and the call is not acknowledged immediately. MASCOT itself does not provide any means by which an application program can react to failure situations itself, unless an IDA's access procedure delivers some kind of error message as one of its parameters and this error message can be set and/or used by the underlying runtime environment.

Now assume that an Activity has called the access procedure of a remote IDA successfully. The Activity will be blocked by the respective handler. After this event the remote node may crash. There is no way to reactivate the blocked Activity again, because the calling node has no need to determine, whether the remote node is still working or not. A solution might be that the handler which in fact issued the remote invocation tries from time to time to contact the remote node. This kind of polling, however, puts some seemingly superfluous load onto the communication network. Nevertheless, it is the only possible way to detect failures in distributed environments. However, the MASCOT philosophy does not provide any help in passing such information to the application program directly.

Similar problems arise, if it is found that a communication link has broken. In this case the underlying runtime environment may try other communication paths to reach the called node.

4.2.2.3 MASCOT Primitives

The MASCOT primitives JOIN, LEAVE, STIM, WAIT, and WAITFOR provide means by which critical regions can be implemented (refer to [2]). They should only be used within the access procedures of IDAs. Access procedures are logically part of IDAs. However, they are called by Activities and are executed as part of the calling Activity, because IDAs are passive elements. The use of the primitives has a direct influence on the schedulability of the Activity. In a distributed environment, however, access procedures are called by a handling instance of an Activity, while the Activity itself is blocked (refer 4.2.1). The primitives therefore only affect the schedulability of the handling instances.

The primitives provide for operations on special objects, called control queues, which behave like semaphores. By calling the primitives Activities either are allowed to continue execution, or are blocked, or release another blocked Activity. Apart from delays imposed by the communication network the application program operates in a distributed system as in a non-distributed system, and no difference can be detected from the outside.

The primitive WAITFOR may cause some unintended effects, because its execution is connected with time considerations. The call of the primitive is supplied with a parameter which defines a time delay. An Activity which issues a WAITFOR instruction is prevented from continuing its processing until a corresponding STIM has been applied or the specified time delay has expired, whichever is the earlier. The reason for the Activity being released from waiting may, optionally, be indicated through the time delay parameter using the following values:

0 to indicated time expired

any other value to indicate STIM received.

The WAITFOR primitive was included into MASCOT to prevent time critical Activities from waiting for ever, if it is impossible for the system to issue any associated STIM operation. In distributed environments delays imposed by the communication network are not covered by using the primitive WAITFOR. Only the handling instance in the remote node benefits from the use of the primitive. A non-determinable delay must be accepted until the calling Activity is released. Nevertheless, if looked upon the application program from outside the distributed environment, the semantics of the WAITFOR primitive are kept. However, if the delays caused by the communication network cannot be accepted by a particular application, the WAITFOR primitive is inappropriate across node boundaries.

To gain the full benefit of the WAITFOR primitive would mean to split an IDA so that its data area resides in one node and the access procedures are copied and loaded in those nodes where they are needed. This approach, however, would imply the implementation of a network-wide synchronization mechanism for control queues what is very complicate and time consuming. The retention of the original semantics of the WAITFOR primitive would therefore not be preferable to its loss.

4.2.3 Solution for MASCOT-like Systems

The algorithm specified in 4.2.1 allows for transfer of control from one node of a distributed system to another node, but logically still executing the same Activity. The system behaves in the same way, if looked upon from outside. The semantics of the MASCOT primitives are kept (in the case of the WAITFOR primitive only, if delays imposed by the communication network are not considered). However, to be able to cope with failures and uncertainties of the distributed hardware environment, at least three facilities should be added to MASCOT:

(1) time-out facility

A time-out facility would allow the specification of the time frame within which a remote node must acknowledge a request.

(2) exception handling facility

An exception handling facility could inform (and thereby release) a blocked Activity, after the call of an IDA's access procedure has failed.

(3) polling facility

A polling facility would allow an Activity (or the associated handler) to check permanently (or from time to time) whether a remote call of an IDA's access procedure is still pending.

Without these facilities MASCOT-like application programs can be executed with only limited success. The underlying runtime environment must handle all error conditions and decide, whether it should abandon the execution of parts of the application program and restart them again after reloading the respective units in different nodes or whether it should ask for user interaction.

4.3 Communication in Ada Systems

Ada programs are distributed on the basis of task objects. The Ada language provides two means by which task objects can communicate with each other: common data areas (global objects) and task entries. Global objects provide for hidden communication links. Task entries specify direct communication links between tasks.

Common data areas may only be accessible by using subprograms (e.g. if the data area is encapsulated in a package). Therefore subprogram calls are to be considered as additional means of communication.

Ada defines different semantics for the three communication mechanisms involved.

4.3.1 Global Objects

In a distributed environment the values of global objects are obtained and updated by use of the communication network. Therefore global objects and the

program locations, at which they are used, must be marked. The address of a global object consists at least of the name of the node, in which the global object resides, and the respective memory location. Every node contains a handler which provides appropriate facilities to access global objects. Because of this provisions all accesses to global objects can be synchronized easily. However, performance penalties may have to be paid.

Ada allows the use of global objects. However, if tasks share objects, it is the responsibility of the programmer to guarantee the integrity and consistency of the shared objects. The underlying runtime environment need not provide synchronization mechanisms for global objects (Because of security and safety reasons at least the compiler should flag the use of global objects in distributed environments). Nevertheless Ada offers a facility, the pragma SHARED, to declare objects to be shared. Accesses to such objects are automatically synchronized by the underlying runtime environment.

4.3.2 Subprogram Calls

Calls of subprograms, which reside in remote nodes, can easily be achieved by the mechanism of remote subprogram calls. The mechanism itself can be implemented similarly to the one described in 4.2.1. Faults arising during the execution of the remote subprogram can be reported to the caller by appropriate exceptions which can then be handled by the caller. The exceptions are to be implementation defined. They should refer to faults of the communication network, faults of the remote node, etc.

Ada does not define explicitly how subprogram parameters are passed. The language allows for the implementation of calls by value and calls by reference. However, an application program must not rely on a specific mechanism. Therefore an actual implementation could facilitate the copy mechanism only, however, with two exceptions: the passing of access objects and the passing of task objects. The following recommendations are made for the implementation of these two cases:

> If a task object is a formal parameter of a subprogram, only means to enable calls of its entries are passed by an actual call. If an entry call is issued, it will be handled as described in 4.3.3.
>
> If an access object is a formal parameter, the use of this object initiates the mechanism provided by the runtime environment to enable the use of objects residing in remote nodes.

The types of formal subprogram parameters are specified by the program text. Therefore the compiler can already establish the respective mechanisms.

4.3.3 Entry Calls

Entry calls establish mechanisms for bidirectional message passing between task objects. The called task executes an optional sequence of statements, while the calling task is blocked. Because task entries can specify a list of formal parameters like subprograms and are associated with a piece of code defined by the accept statement, entry calls are treated as remote subprogram calls. However, tasks calling a specific entry are synchronized on a first-come

first-served basis, wheras subprograms are re-entrant.

To prevent tasks waiting forever for the return from an entry call, Ada defines two special types of entry calls: conditional entry calls and timed entry calls.

A conditional call should be issued, whenever a calling task cannot wait for the execution of the associated rendezvous. If the rendezvous cannot be performed immediately, the entry call is abandoned and the calling task executes a specified sequence of statements. In a distributed environment the semantics can be defined as follows. An appropriate request is transmitted to the remote node. The remote node either processes the rendezvous or answers that the rendezvous is impossible at the moment, because the called task does not wait at an associated accept statement. The Ada semantics are not changed by this solution. The delay imposed by the communication network, however, must be considered.

A timed entry call should be issued, whenever a rendezvous must be started within a certain period of time. The calling task specifies the possible delay with the entry call. If the rendezvous is not started within the specified period of time, the entry call is abandoned and the calling task executes an optional sequence of statements. The defined semantics can be achieved in a distributed environment as follows. The calling node issues an appropriate request for the remote node. If the request is not answered within the specified delay, an appropriate message is transmitted to the remote node and the entry call is abandoned, unless the remote node states that the rendezvous is in progress. This solution is in accordance with the Ada language definition.

4.4 The Communication Protocol

All communication transmitted via the communication network is handled by a communication protocol. The protocol selects the proper communication paths between nodes according to the requirements of Activities (in MASCOT-like application programs) or of task objects (in Ada programs).

The protocol detects failures in the communication network and crashes of nodes. If possible, messages are routed via third nodes in the case of faults. Proper error information is passed to the caller.

Considerations concerning the actual implementation of a communication protocol are beyond the scope of this paper.

5. The Runtime Environment and Associated Tools

An application program running on a distributed environment requires a different underlying runtime environment to a program running on a single processor system. Some aspects of this special runtime environment have already been discussed in previous chapters. Compilers, linkage editors, and loaders must be aware of the special interface of the runtime environment.

A requirement for the automatic distribution process is that a programmer should not be aware of the target environment and of the actual distribution of his program onto the target environment. An application program will be split according to a special strategy and to constraints imposed by the target environment. Necessary connections to the underlying runtime system are made to suit the particular distribution. The distribution process must inform the code generation phase of the compiler, the linkage editor, and the loader accordingly.

5.1 Ada Programs

As discussed in 4.3 the semantics of Ada are not affected by the necessary communication between the nodes of a distributed system. An Ada program can therefore be developed independently of a target environment (either single computer system or distributed system), unless machine dependent features are used in the program text or special implementation defined exceptions (e.g. failures of the communication network) are handled by the program itself.

The runtime system implied by the language definition for a single computer system must only be extended by the following features:

> means by which objects residing in remote nodes are accessed (possibly including synchronization of the requesting task objects);
>
> means by which subprograms residing in remote nodes are invoked (including proper transfer mechanisms for the subprogram parameters);
>
> means by which the entries of task objects, which reside in remote nodes, are called (including proper transfer mechanisms for the entry parameters);
>
> means by which tasks residing in remote nodes are initiated and by which their termination is told the initiating node.

A powerful communication system must be associated with these features. The communication system should detect failures of the communication network (and possibly correct them itself, e.g. by using another communication link) and failures of nodes (e.g. crashes). All faults should be reported to the application program via proper implementation defined exceptions. At least the following exceptions should be provided: fault_of_communication_link, time_out, and failure_of_remote_node. The need for those exceptions has been discussed in 4.2.2.2. The exceptions would allow an application program to perform some kind of error recovery. Because task are objects of an Ada program itself and not of the runtime environment, the only error recovery which can be done by the runtime environment is to try redundant communication links between nodes, if a particular link is broken.

An Ada program always starts its execution with that procedure which was

esignated as main program in the usual sense. Therefore a distributed nvironment will commence work with that node which hosts the main program. The ther nodes will start when the tasks they host are initiated. A special tart-up facility is not necessary.

he distribution process computes a scope tree with use and call relationships rom the application program. A tree-like representation of an Ada program xists somewhere during its compilation (normally after the analysis phase of he compiler). This tree-like representation already contains almost all nformation needed by the distribution process. The synthesis phase (code enerator) of an Ada compiler should know the distribution considerations made y the distribution process to be able to generate proper code and to establish roper connections to the runtime environment. The distribution process herefore could be a third phase of an Ada compiler which reshapes the program ree computed by the analysis phase to suit the distribution decisions.

he linkage editor and the loader must be able to use tables produced by the istribution process to bind and load the application program according to the istribution considerations.

.2 MASCOT-like Programs

he MASCOT philosophy aims at the development of application programs intended o run on single computer systems. Nevertheless the incorporated design ethodology eases the splitting of MASCOT-like programs, because they are epresented by a graph: the ACP Diagram. However, MASCOT also defines a runtime ernel for the execution of application programs. The kernel offers ynchronization primitives based on semaphores (in MASCOT called control ueues). This approach is not suitable for distributed systems, because it equires network-wide synchronization of accesses to control queues and this is ery hard to achieve needing a lot of organisational overhead throughout the ystem.

possible solution to the problem raises the level of synchronization up to the alls of access procedures of IDAs, treating the access procedures as monitors. s shown in 4.2.2.3 the intended semantics of the primitive WAITFOR are lost hen. Also failures of the communication system and of remote nodes cannot be andled by the application system itself anymore, unless an exception handling acility and some other features as discussed in 4.2.3 are added to MASCOT.

ystem recovery can be achieved very easily after failures in MASCOT-like pplications, because Activities are totally independent of each other. If a ode crashes, the respective Activities need only be reloaded in other nodes and estarted. The reloading of IDAs is more difficult, as their data areas should e restored. Nevertheless recovery can be performed without interfering with he application program.

MASCOT Machine defines facilities with which the execution of the application rogram can be monitored online. These facilities will add some load on the ommunication network. A designated node must act as an interface to allow user nteraction. Monitor requests will be sent from this node to other nodes. The onitor facility needs an image of the distributed application program to be ble to determine the locations of all program parts.

MASCOT Frozen Machine can start, halt, resume, and terminate Subsystems. To rovide this service facilities similar to the monitoring facilities must be

incorporated into the runtime environment. A MASCOT Evolutionary Machine allows alteration of an application program online. Such facilities make an automatic distribution process obsolete.

Linkage editors and loaders are driven by a table generated by the distribution process.

Conclusion

decrease software life-cycle costs application programs should be developed
ependently of target environments. Recent years have shown a trend to use
e and more distributed hardware systems in realtime environments to achieve
ater flexibility and reliability. To be able to keep application programs as
lependent as possible from the distributed target environments methods must be
ınd to distribute application programs automatically over target environments
. keeping realtime constraints and efficiency.

.s study has shown that automatic distribution is possible. A distribution
›gram only needs graph-like representations of both application program and
-get environment. The task of the program then is to map the graph
›resentation of the application program on that of the target environment
ısidering several constraints such as input/output constraints. The
ıstribution strategies, however, are based on rather heuristic assumptions and
ısiderations. More research is necessary in this area. This study could only
:line the very basic concepts of a distribution process and some aspects of
› necessary runtime environment and associated tools. However, it has been
ınd that automatic distribution is feasible.

e semantics of the Ada language are adequate in distributed environments.
ly some implementation defined exceptions should be added to be able to handle
ror conditions special to distributed environments on application program
vel. The call and use relationships introduced in this paper are also useful
analyse data and control flow within an application program and to derive
timization criteria.

SCOT-like programs seem to be distributable very easily because their
velopement is based on a graph representation: the ACP diagram. The
ecialities of the MASCOT Machine (Frozen or Evolutionary), however, complicate
e distribution process. Moreover, the absence of exception handling
cilities in the MASCOT philosophy beg for amendments of the runtime kernel.
ditionally the semantics of the primitive WAITFOR are not entirely
propriate. The possibility to influence from outside the system the
nedulability of a number of Activities (subsumed under a Subsystem) adds a
nsiderable runtime overhead to the system. The features of an Evolutionary
SCOT Machine which allow the changing of the whole application program online
e in contradiction to an automatic distribution process. Therefore, in the
se of an Evolutionary MASCOT Machine, hand distribution of application
ograms is recommended. Because of the monitoring facilities of MASCOT
chines a node of a distributed environment must be designated which allows
er interaction with the distributed application program as a whole.

an overall conclusion it must be stated that application programs which are
pposed to run on distributed environments must be developed on host computers.
l tools necessary for an efficient distribution of the application program
st be available on the host computer.

7. Acknowledgements

This study was carried out while the author was working in the Computing Division, RSRE on attachment from the Federal Republic of Germany.

The author thanks F. R. Albrow from RSRE for his great help in compiling this paper. Without his suggestions and corrections this paper would never have been completed.

8. References

[1] Reference Manual for the Ada Programming Language
ANSI/MIL-STD 1815 A
January 1983

[2] Joint IECCA & MUF Committee on MASCOT
The Official Handbook of MASCOT
MASCOT II Issue 2
March 1983

[3] C1719 - Further Ada Studies
Use of MASCOT in the Mapse
Document No: C1719/REP/10 issue 1
August 1982

[4] Systems Designers Limited
Multiprocessor MASCOT
Final Report on Contract A71C/6021
Document No: C1394 issue 1
August 1982

[5] Gustav Fickenscher
The Use of the MASCOT Philosophy for the Construction of Ada Programs
R.S.R.E. Report No. 83009
October 1983

[6] W. R. Franta, W. E. Boebert, H. K. Berg
An Approach to the Specification of Distributed Software
in
H. K. Berg, W. K. Giloi (Ed.)
The Use of Formal Specification of Software
June 25-27, 1979, Berlin
Informatik-Fachberichte, Vol. 36
Springer-Verlag Berlin Heidelberg New York 1980

[7] F. R. Albrow
MOD (PE) R.S.R.E., St. Andrews Road, Great Malvern, Worcs. WR14 3PS
Private Communications
1983/1984

[8] G. Goos
Uebersetzerbau (Compiler Construction)
Lecture Script (in German)
University of Karlsruhe, Institut fuer Informatik II

DOCUMENT CONTROL SHEET

Overall security classification of sheet .....UNCLASSIFIED.....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference	3. Agency Reference	4. Report Security Classification
	MEMORANDUM 3696		UNCLASSIFIED

5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location
	ROYAL SIGNALS AND RADAR ESTABLISHMENT, MALVERN

5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location

7. Title
AUTOMATIC DISTRIBUTION OF PROGRAMS IN MASCOT AND ADA ENVIRONMENTS - A FEASIBILITY STUDY

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp.	ref.
FICKENSCHER, G			10/5/84	37	8

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement
UNLIMITED

Descriptors (or keywords)

MASCOT
Ada
Distributed Systems
Automatic Software distribution

continue on separate piece of paper

Abstract Recent years have seen a steady increase of computer systems based on distributed hardware. This is made possible by the reduction of hardware costs and increases in the power of hardware components. More sophisticated software systems are the result. To reduce the costs for software development it would be advantageous to reuse software systems in different target environments. In the case of distributed systems this means that software may be distributed differently in different environments. It would simplify matters, if the software could be distributed automatically. In this paper the feasibility of automatic distribution of Ada programs and MASCOT-like programs is investigated. The impacts of the distribution on the runtime environment and communication system are outlined.

END

FILMED

5-85

DTIC