

AD-A137 417

THE USE OF THE MASCOT PHILOSOPHY FOR THE CONSTRUCTION
OF ADA PROGRAMS(U) ROYAL SIGNALS AND RADAR
ESTABLISHMENT MALVERN (ENGLAND) G FICKENSCHER OCT 83

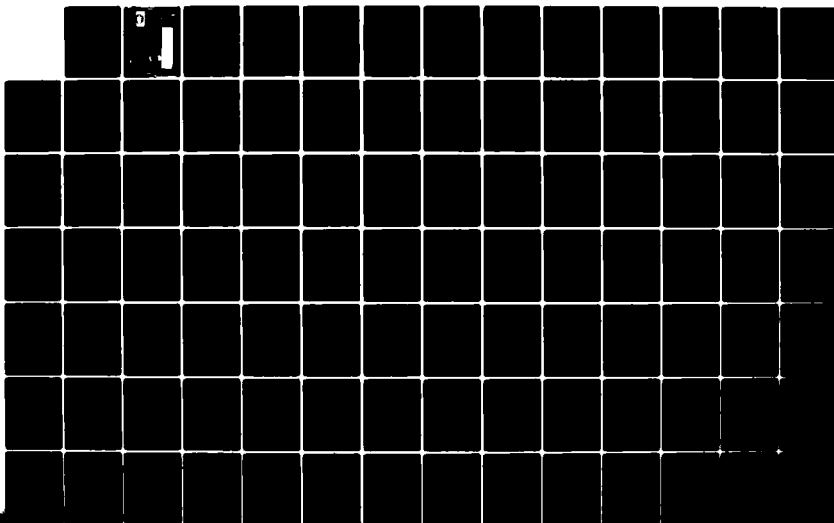
1/2

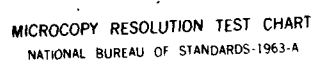
UNCLASSIFIED

RSRE-83009 DRIC-BR-90207

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A 137417

THE USE OF THE SACRED SYMBOLS IN THE CONSTRUCTION
OF THE UNIVERSE

Author: C. F. F. F.

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Report No 83009

TITLE: The Use of the MASCOT philosophy for the construction of Ada programs
 AUTHOR: FICKENSCHER, G
 DATE: OCTOBER 1983

↓
 The development of computer based systems poses major problems on the people involved. Both, MASCOT (the official design methodology of the UK Ministry of Defence for real-time systems) and Ada (the official programming language of the US Department of Defence for embedded computer systems) claim to offer a solution to the majority of these problems. MASCOT is a programming support environment which is independent of a particular programming language, but it defines its own runtime kernel for parallel execution of different program parts. Ada, on the other hand, offers language constructs to express parallelism of program parts, but Ada enforces a particular design methodology with its language rules.

This paper investigates whether it is feasible to combine the MASCOT methodology with the programming language Ada. It demonstrates a possible implementation of a MASCOT Construction Data Base in Ada, and it explains the combination of MASCOT and Ada by using a simple example.



Accession For	
NEIS	<input checked="" type="checkbox"/>
DCS	<input type="checkbox"/>
Under	<input type="checkbox"/>
Justified	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

Copyright
 C
 Controller HMSO London
 1983

CONTENTS

1. Introduction	1-1
2. Structure of the Document	2-1
3. Evaluation of MASCOT with Respect to Ada	3-1
3.1 MASCOT Overview	3-1
3.2 Ada Overview	3-2
3.3 The MASCOT Machine	3-5
3.3.1 Constructing	3-7
3.3.1.1 Production of System Element Templates	3-7
3.3.1.1.1 IDA Templates	3-8
3.3.1.1.2 Root Templates	3-9
3.3.1.2 Creation and Destruction of System Elements	3-9
3.3.1.3 Forming and Deleting Subsystems	3-10
3.3.1.4 Command Interpreter	3-10
3.3.2 Controlling	3-11
3.3.3 Scheduling	3-11
3.3.3.1 Scheduler	3-11
3.3.3.1.1 Adjustments to the Priority of an Activity	3-11
3.3.3.1.2 The Selection of an Activity at the Start of a Slice	3-11
3.3.3.1.3 The Duration of a Slice	3-12
3.3.3.1.4 End of Slice Action	3-12
3.3.3.2 Synchronisation	3-13
3.3.3.3 Timing	3-13
3.3.3.4 Activity Suspension and Termination	3-13
3.3.4 Device Handling	3-13
3.3.5 Monitoring	3-14
3.3.6 Interacting	3-14
3.4 Activities	3-15
3.5 Inter-Communication Data Areas	3-15
3.6 Subsystems	3-16
3.7 Frozen and Evolutionary Systems	3-17
3.7.1 Frozen Systems	3-17
3.7.2 Evolutionary Systems	3-18
4. A Solution	4-1
4.1 The Construction Data Base	4-1
4.1.1 Data Structure of the MASCOT Development System	4-2
4.1.1.1 Basic Types	4-2
4.1.1.2 Type Summary	4-2
4.1.1.2.1 Types of Entries	4-2
4.1.1.2.2 Parameter Types	4-3
4.1.1.3 List Types	4-3
4.1.1.4 The Types of the Parameters	4-4
4.1.1.5 The Type "MASCOT_System"	4-5
4.1.1.6 The Type "subsystem"	4-5
4.1.1.7 The Type "activity"	4-6
4.1.1.8 The Type "IDA"	4-6
4.1.1.9 The Type "root_template"	4-7
4.1.1.10 The Type "IDA_template"	4-7
4.1.1.11 The Type "data_type"	4-8
4.1.2 The ACP Diagram	4-8
4.1.2.1 The Type "node"	4-9
4.1.2.2 The Type "arc"	4-9
4.1.2.3 The Construction Procedure for the ACP Diagram	4-10
4.1.3 Creation of Data Types	4-10
4.1.4 Creation of IDA Templates	4-11
4.1.5 Creation of Root Templates	4-11

4.1.6 Forming of Subsystems	4-11
4.1.7 Forming of the Main Program	4-12
4.1.8 The Main Program of the Construction Process	4-13
4.2 Data Types of the IDAs	4-15
4.3 The Form of IDA Templates	4-16
4.3.1 Channel Template	4-16
4.3.2 Pool Template	4-17
4.4 The Form of Root Templates	4-17
4.5 Implementation of Roots and IDAs	4-18
4.5.1 Roots	4-19
4.5.2 IDAs	4-19
4.6 Creation of IDAs	4-21
4.7 The Form of Subsystems	4-22
4.8 The Form of the Main Program with Subsystems	4-24
4.9 The Form of the Main Program without Subsystems	4-25
5. Final Remarks	5-1
6. References	6-1
APPENDICES	
Appendix 1: Programs for Construction of Data Base	
Appendix 2: An Example	

1. Introduction

The development of complex computer based systems poses major problems to the people involved. MASCOT (Modular Approach to Software Construction Operation and Test) serves as both a methodology and a tool set to overcome managerial and technical difficulties. MASCOT is claimed to be independent of programming languages. However, MASCOT provides a runtime kernel and its use implies several restrictions on the programming language used to implement a particular computer based system.

MASCOT presents no problems if used in conjunction with a programming language that does not have built-in tasking facilities. As the example of CORAL 66 shows, a language may have to be extended to meet the requirements of MASCOT but the extensions do not influence the semantics of the sequential part of the language. The compiler of the "extended language" can only be used in a MASCOT environment.

Ada must not be changed in any way (refer [2]) and Ada provides its own tasking facilities which cannot be superseded by the semantics of a given runtime kernel. However, an Ada program is not required to contain Ada tasks. Thus, it is possible to provide an alternative runtime environment that supports concurrency with Ada program units other than Ada tasks (refer [2], subchapter 1.1.1 (i)) provided that the program units themselves conform with the language rules. This means that the program units must be main programs in the Ada terminology. The resulting MASCOT system is similar to that using CORAL 66.

Nevertheless, it is worth-while to investigate whether it is possible to utilize the complete Ada language, which is intended for writing large systems, in conjunction with the MASCOT philosophy, which is proved to be a reasonable design instrument for large systems. This paper offers a solution to the problem.

2. Structure of the Document

In chapter 3 MASCOT is evaluated with respect to Ada. The evaluation is made taking initial thoughts and solutions from [3].

An initial implementation of a MASCOT Machine in an Ada environment is given in chapter 4. The solution, however, only considers a Frozen System and is to be looked at as a proposal. It is not tested because of the non-availability of an Ada compiler. The related Ada programs are listed in Appendix 1.

Chapter 5 serves as conclusion.

The use of the MASCOT philosophy in an Ada environment is demonstrated by a simple example. Its detailed description and the source codes of the various program units are given in Appendix 2.

3. Evaluation of MASCOT with Respect to Ada

3.1 MASCOT Overview

A full description of the MASCOT approach is neither possible nor desirable here, for that the reader is referred to [1]. It is, however, worth outlining the basic ideas of the MASCOT concepts for the reader who is not already familiar with MASCOT.

The central idea of MASCOT is to handle the implementation of complex realtime systems by providing a method to deal with the concurrency. This method decomposes the system into a set of sequential processes (called "activities") each of which operates in parallel with the others, none having knowledge of the others.

Data to be shared between Activities is placed in Inter-Communication Data Areas (IDAs). Activities are not allowed to access the data directly. Instead a set of access procedures is specified for each IDA, and Activities call these procedures. The procedures resolve contentious concurrent accesses by implementing critical regions, which provide mutual exclusion. MASCOT provides several primitives in its runtime kernel to implement the critical regions.

MASCOT defines two kinds of IDAs:

(a) Channels

A Channel is used to pass data in the form of messages between Activities. A Channel has two unidirectional interfaces: an input interface used by Activities that produce messages, and an output interface used by Activities that consume messages. There can be an accumulation of unconsumed messages within a Channel. A Channel might provide additional interfaces for monitoring purposes.

(b) Pools

A Pool holds data for reference purposes. A Pool can have many different interfaces for many different purposes.

Only the designers of IDAs are concerned with parallel execution. The designer of an Activity is concerned with the sequential operations of the Activity itself and only needs to know the procedural interfaces of the respective IDAs and the structure of their data. This approach facilitates the testing of Activities in a test harness.

Because an I/O device either produces data or consumes data it can be connected to an Activity via a Channel. Conceptually devices can be looked at as special Activities. The runtime kernel allows to connect interrupts to devices.

MASCOT does not specify how an overall system is to be decomposed into Activities and IDAs. It provides a notation for expressing the design in the form of an Activity Channel Pool (ACP) Diagram.

The method by which the application system is constructed is defined by MASCOT. Recognising that systems will often contain several essentially identical Activities or IDAs, "templates" are used for the code of Activities (called "Root templates") or IDAs. Several instances can then be created from each template.

The correct connectivity of the system is checked when it is built from Root instances and IDAs. A Root template specifies the IDA templates it operates on.

To allow a hierarchical composition of larger systems Activities and IDAs can be subsumed to a "Subsystem". Then, some of the IDAs are private to the Subsystem. Others form the interfaces to other Subsystems and will be referred to as "Subsystem-IDAs" (called SIDAs). A Subsystem is allowed to consist of one Activity only. Thus, from a more general point of view, an application is built from Subsystems.

MASCOT defines two modes of construction: "frozen" and "evolutionary". Construction of a frozen system is equivalent to conventional system building. An evolutionary system is one where construction (and dismantling) can occur in the system while it is running. Only Subsystems can be formed, started, stopped, and removed online.

MASCOT defines a runtime kernel which provides a set of primitives to the application system. These are concerned with synchronisation, mutual exclusion, timing, scheduling, device handling, controlling Subsystems and monitoring. A particular MASCOT development system might not support all these facilities. A minimum compulsory subset, however, is defined.

The essential elements of MASCOT are:

- (a) a method of decomposing a system into independent, in parallel executable parts with procedural interfaces, and a notation (the ACP diagram) which the application system can be derived from. This method incorporates I/O devices easily.
- (b) a method of construction involving templates and various defined checks.
- (c) a definition of a runtime kernel that supports the application system.

3.2 Ada Overview

For a full description of the Ada programming language the reader is referred to [2]. It is, however, worth outlining some of the basic ideas of Ada for the reader who is not familiar with Ada.

Ada was developed for the US Department of Defense to meet the urgent need for reliable implementations of embedded computer systems. The user of Ada thinks in terms of data types and operations on these types during the implementation of his particular system and not in terms of sequential functions. Moreover, the idea that a system should be composed of almost independent parts (modules) led to a concept of extensive modular programming in Ada.

Ada's package concept was derived from the ideas of abstract data types and extensive modularisation. A package consists of a well-defined interface (specification) visible to the user of the package and of an implementation part (body) hidden from the user of the package. The designer of the package can restrict the use of objects derived from visible data types.

Packages can form all or part of a project library as can subprograms (specification and/or body), generic units, and instances (instantiations) of generic units. Subprograms are procedures and functions in the usual sense.

Library units are compiled separately.

The visible parts (interfaces) of library units can be imported by compilation units by naming the units needed in a special clause, called context clause, at the start of a compilation. The compiler then ensures that the interfaces are strictly obeyed by the user.

Because of the strong typing incorporated in Ada a method was introduced to parameterise packages and subprograms. Such packages and subprograms are said to be generic and are really templates from which actual units can be derived by instantiation. Possible parameters are data objects, data types, and subprograms. It is possible to restrict the use of objects of given generic data types from within the generic unit itself. An instantiation provides the actual parameters.

Libraries provide for the bottom-up development of software systems written in Ada. To provide a top-down approach also, Ada allows a user to specify subunits. These subunits are not library units. Therefore the specifications of the respective "visible" parts must be given within library units (parent units) and their bodies must be declared as separate from the parent units. The bodies are then compiled separately and must mention their parent units. Possible subunits are the bodies of subprograms, of packages, and of tasks. Subunits have direct access to all objects declared in or visible to their parent units.

The compilation of units must follow a set of rules. Specifications have to be compiled prior to their bodies. Parent units have to be compiled prior to their subunits. Imported library units have to be compiled prior to those library units mentioning them in a context clause. If a compilation unit is recompiled, all dependent units must be recompiled.

Because of Ada's commitment to abstract data types tasks are treated as data types with certain restrictions. A task is specified either as a task object, then only one task object of an anonymous task type is created, or as a task type, when several task objects of the same type can be declared. A task body is associated with each task (type) specification. The body defines the sequence of statements executed by the respective task object.

A task object is implicitly activated at its point of declaration. Tasks are terminated, when they have executed their last executable statement and no dependent task is still active, or, when they have selected a terminate alternative in a selective wait statement and the scope in which they are declared is left. Tasks can be explicitly aborted by other tasks via an abort statement.

Communication between tasks is achieved in two different ways:

- (a) tasks communicate through objects which are in their visible scope;
- (b) tasks communicate via entry calls and associated accept statements.

The first approach is very unsafe because concurrent accesses to the objects are not resolved. The language, however, provides a pragma SHARED to identify such shared objects. If an object is marked to be shared, mutual exclusion is implicitly achieved in accessing such an object.

The second approach implements a bidirectional message handling via a mechanism called rendezvous. Tasks specify a set of entries, which can be looked at as messages boxes, in their task (type) specification. If a task wants to deliver

a message to another task, it executes a statement which calls the respective entry. If a task wants to consume a message, it executes an accept statement which names the respective entry declared in its specification. The performance of a rendezvous is an indivisible operation during which the two tasks are said to be in rendezvous.

Three different kinds of entry calls exist:

(a) normal entry call (commonly called an entry call)

A task calls the entry of another task. If the called task is waiting at an associated accept statement, the rendezvous will be performed immediately. Otherwise the calling task will wait.

(b) conditional entry call

The rendezvous must either be performed immediately or abandoned. Only if the called task is waiting at an associated accept statement, the rendezvous will be performed. If the entry call is abandoned, the calling task executes a specified sequence of statements.

(c) timed entry call

The rendezvous must be performed within a given time frame. If the called task fails to arrive at an associated accept statement within the time frame, the entry call is abandoned, and the calling task executes an optional sequence of statements.

Two kinds of accept statements exist:

(a) accept statement

When a task reaches an accept statement, and an entry call is already pending, the rendezvous is performed by executing an optional sequence of statements. Otherwise the task waits until a call of the associated entry happens.

(b) selective wait

A selective wait statement contains at least one accept alternative, which consists of an accept statement. In addition it may contain either a terminate alternative (only one), or one or more delay alternatives, or an else part; these three possibilities are mutually exclusive.

An optional condition is associated with each alternative. If the condition is true, the alternative is said to be open and can be chosen. If there is no condition, the alternative is always open. The way an open alternative is chosen is arbitrary.

The else part is chosen, if no alternative is open.

Delay alternatives are chosen, if an accept alternative cannot be chosen in a given time frame. An optional sequence of statements is then executed.

The terminate alternative can only be chosen, if no other alternative is open and the task is allowed to terminate.

Tasks waiting at an entry are served first-in first-out. Tasks, however, can have a static priority. The priority is not evaluated in the case of a rendezvous. It is, however, possible to implement particular strategies using the conditions of selective wait statements.

It is possible to connect entries to interrupts. This is done by using a so called representation specification for a particular entry. The effect of the connection is left to the particular implementation.

The language does not specify the scheduling of tasks. In particular, the only way to schedule tasks explicitly is by use of a delay statement. The execution of a task is suspended at least for the duration specified by the delay statement.

An implementation of Ada must provide a runtime system to meet the requirements of the semantics of the built-in tasking.

3.3 The MASCOT Machine

The environment which supports MASCOT is a conceptual "MASCOT Machine" on which the facilities needed to develop and run a MASCOT application are available to the user.

In a MASCOT Machine a set of software construction tools builds and maintains a Construction Data Base, from which the application system is constructed ready for execution. The MASCOT Machine may also provide control and monitoring of the system after it has been started.

Figure 3.3-1 shows the several facilities provided by the MASCOT Machine. The facilities fall into a number of groups. The following subchapters describe the various groups and compare them with equivalent Ada features.

An implementation of the MASCOT Machine does not need to provide all facilities. There is, however, a mandatory set of facilities which defines a Minimal MASCOT Machine (refer Fig. 3.3-2).

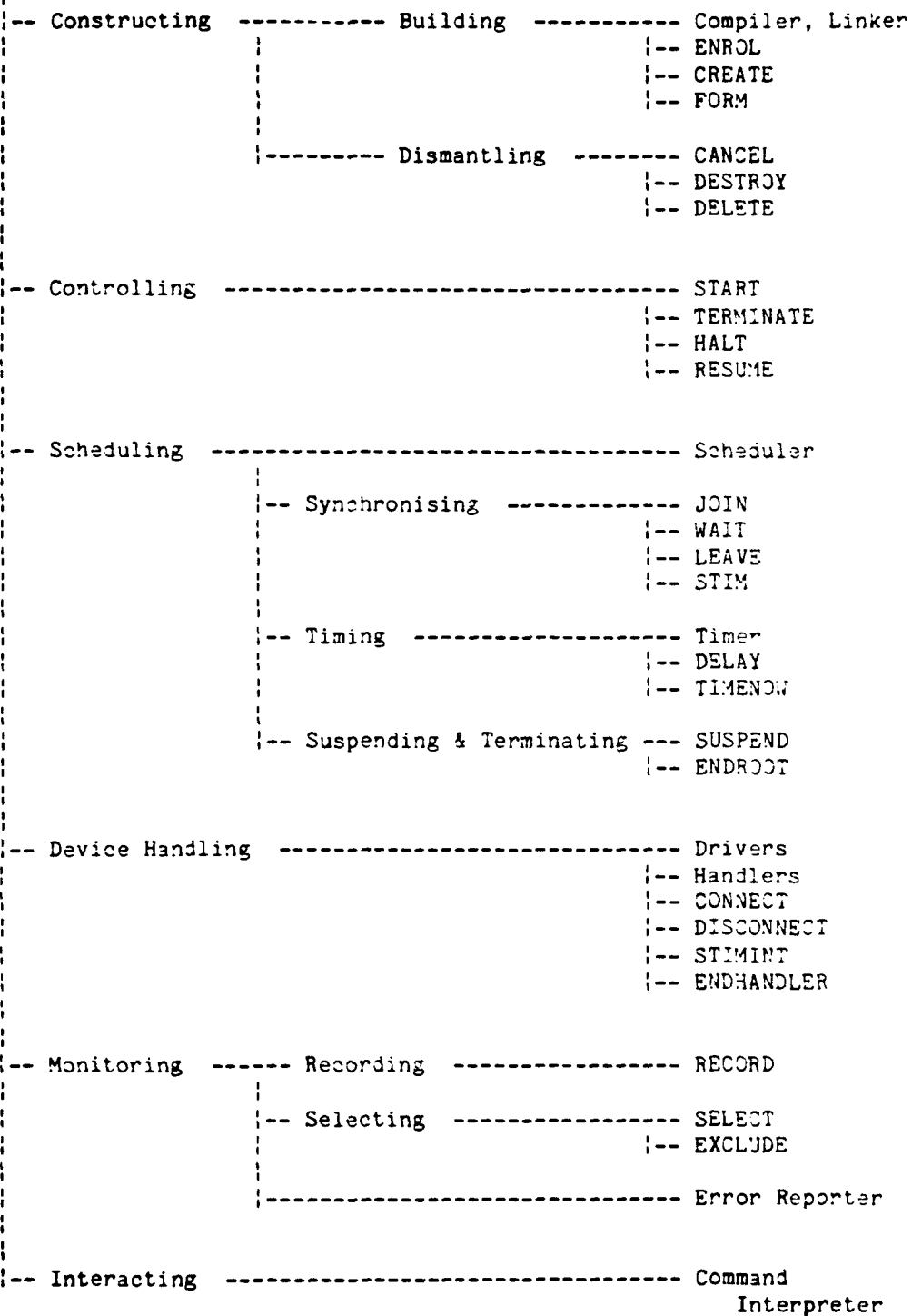


Figure 3.3-1: The MASCOT Machine

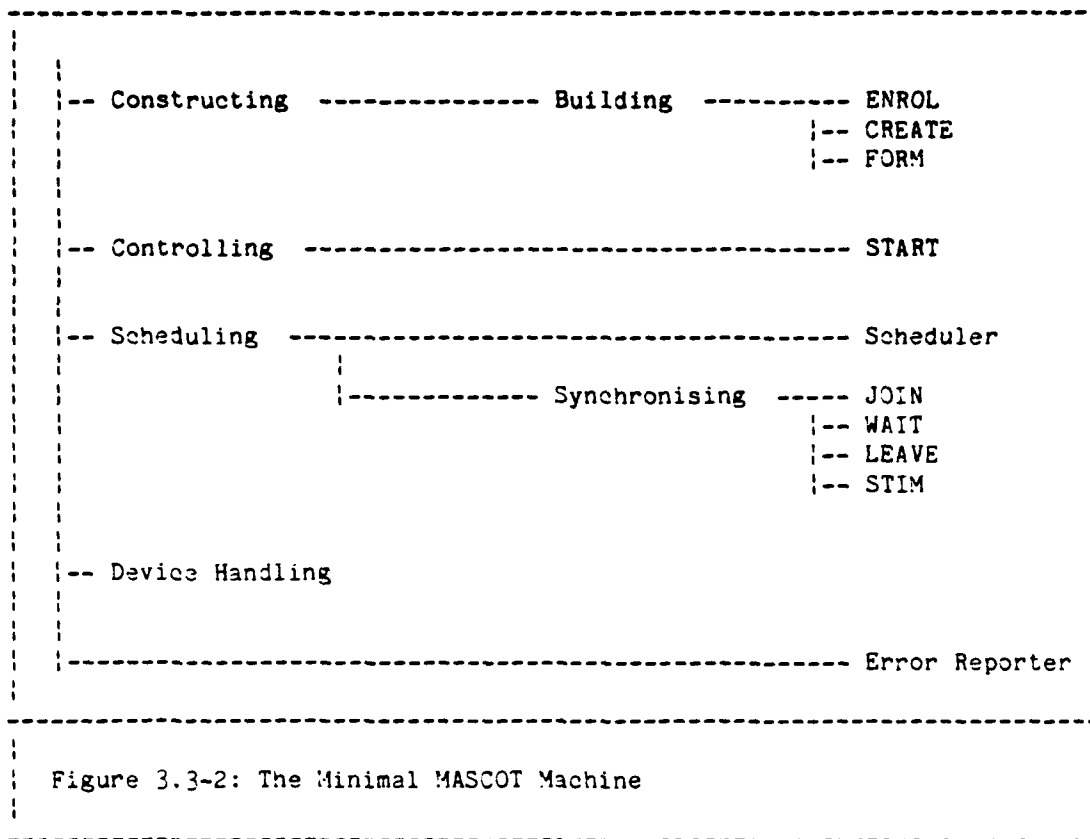


Figure 3.3-2: The Minimal MASCOT Machine

3.3.1 Constructing

Construction is the term used in MASCOT to cover the transition from the detailed design description of a network (normally the ACP diagram) to the implementation of that network as executable software. MASCOT requires that the transition is achieved through operations that are consistent with the modularity defined by the design.

The construction in MASCOT involves three clearly separable stages:

- the specification of System Element Templates (SET) and their enrolment in a Construction Data Base;

- the creation and destruction of System Elements (SE);

- the formation and deletion of Subsystems.

3.3.1.1 Production of System Element Templates

The initial stage in the construction of software in MASCOT is the production of SETs. This requires the MASCOT Machine user to:

write a specification that expresses the internal details of the SET as source text in a programming language;

ENROL the SET in the Construction Data Base, making it known to the MASCOT Machine for use in later stages of construction.

The act of enrolling a SET may involve compilation and link loading of the source text. After enrolment the following information is available in the Construction Data Base:

- the name of the SET;

- its connectivity constraints;

- the places from which constituent parts can be retrieved.

A SET, once enrolled, remains known to the MASCOT Machine and is available for use until it is cancelled when knowledge of it is destroyed by removal from the Construction Data Base. Cancellation of a SET is only allowed if it is not required by another SET. Cancellation is effected by the CANCEL facility.

Ada provides a facility to describe templates, called generics in Ada terminology, of program units. An Ada implementation provides a kind of construction data base, the project library, while an Ada compiler ensures the correct connectivity of program units. MASCOT-like information is stored in the project library. The enrolment of a SET is achieved in an Ada environment by compiling it. Cancellation can only be achieved by using a library user package working on the project library. An entry in an Ada project library is replaced, if a new entry with the same specification characteristics and the same name is compiled.

There are three kinds of SETs in MASCOT: Channels, Pools, and Roots. But only two kinds of SETs are considered by the MASCOT Machine, because Channels and Pools are constructionally equivalent.

3.3.1.1.1 IDA Templates

The information in a SET for an IDA comprises:

- its name and class (Channel or Pool);

- a specification of its data area and the MASCOT control queues which it requires;

- the access mechanisms which define the operations allowed on the data area.

Provisions must be made for initialising IDAs.

The IDA structure can be achieved by using Ada packages. Moreover, this allows easy initialisation. To provide the possibility of implementing different IDAs using the same data types the definition of an IDA should be distributed over two packages:

- a package which specifies the abstract data type used by several IDAs;

- a generic package which defines the data area, the particular SET works on,

and implements the access mechanisms of the particular SET.

3.3.1.1.2 Root Templates

The implementation of a Root is largely concerned with the expression of the required algorithm. Its connectivity constraints must be formally defined in terms of the number and type of the IDAs to which it must be connected when it is used to form an Activity.

Therefore the information in a SET for a Root comprises:

- its name, and the fact that it is a Root template;

- specification of each formal IDA parameter (These are name and class of the particular IDA with naming the access mechanisms needed to access this IDA);

- a specification of any constant value parameters;

- the body of the Root (This is the processing algorithm expressed as a program sequence. It includes calls on the access mechanisms specified in the formal IDA parameters).

A Root may be a substantial program with an own internal structure. The MASCOT Machine only constrains the interfaces needed.

In Ada a Root template can be expressed as a subprogram. The data types of the connected IDAs must be made visible to it. The access procedures to the IDAs can be formal in terms of generic parameters. The implementation of the IDAs and the IDA access procedures not needed are hidden from the Root template.

3.3.1.2 Creation and Destruction of System Elements

Users of a MASCOT Machine must be able to create SEs necessary for the Subsystems they wish to construct. The CREATE facility is handed the identity of an enrolled SET and the name of the SE required. CREATE brings a Root or IDA into existence using the SET, and may apply any IDA presets or initialisation procedures at the same time. Information on the newly created SE is held in the Construction Data Base for use in later stages of construction.

Using Ada, CREATE can be achieved by instantiating the generic template of a SET. The SE is then held as a compilation unit in the project library. Because of Ada's visibility and compilation rules and because of Ada's tasking concept the creation of IDAs private to Subsystems and the creation of Activities are to be performed together with the formation of Subsystems.

A user may be allowed to destroy SEs which have previously been created. This is effected by the MASCOT facility DESTROY. The destruction, however, is conditional upon the named SEs not being in use in a formed Subsystem. In an Ada environment this facility can only be provided, if a user package working on the project library exists or if a SE is recompiled.

3.3.1.3 Forming and Deleting Subsystems

The final stage in the construction of software in MASCOT is the connecting of SEs which have been previously created to build executable software whose modularity is consistent with the ACP diagram. This is the FORM process and the network fragment formed in a single operation is called a Subsystem.

The information required by the FORM facility is:

- the name of the Subsystem to be formed;

- a list of Activities, each Activity is supplied with the following information:

 - the name of a previously created Root which is not already in use;

 - a list of previously created IDAs which satisfy the formal parameter requirements of the respective Root template;

 - its actual value parameters.

A Subsystem is in the "idle" state after formation.

When a Subsystem is started, all its Activities are activated in parallel. In an Ada program this effect can be achieved by calling a procedure which has local tasks. After entering the called procedure all its local tasks are activated in parallel. Therefore in an Ada environment a Subsystem is a procedure with Activities as local tasks. A Subsystem can be tested independently, if a test harness is provided which calls the procedure denoting the Subsystem and produces or consumes data sent through the Subsystem-IDAs needed by the Subsystem.

IDAs private to a Subsystem need not be visible outside the Subsystem. They are local packages of the procedure denoting the Subsystem. Therefore the formation process must distinguish between private IDAs and Subsystem-IDAs. Activities and private IDAs are created together with the formation of their Subsystem.

An important aspect of the Subsystem formation is the ability to check that actual connections defined by each Activity do not violate the connectivity constraints in the appropriate SETs. This is achieved easily in an Ada environment because the compiler will perform all these checks.

Users may be allowed to delete Subsystems, which they have previously formed, with the MASCOT facility DELETE. Only Subsystems in the "idle" state can be deleted. Deletion can only be achieved by recompiling or by a library user package working on the project library in an Ada environment. The constituent SEs are not destroyed in either environment.

3.3.1.4 Command Interpreter

A MASCOT Machine may make the construction operations available either through a procedural interface or through the Command Interpreter in an evolutionary system.

An Ada Programming Support Environment (APSE) using the MASCOT philosophy must

provide similar tools. A Command Interpreter, however, is part of an APSE. It is to be amended to meet the requirements of MASCOT.

3.3.2 Controlling

A MASCOT Machine provides a set of facilities for controlling the execution of a Subsystem. These facilities start, halt, resume, and terminate execution. They are mandatory in evolutionary implementations but optional in frozen systems.

These functions are not applicable to an Ada environment, because tasks are started implicitly at their point of declaration and cannot be halted or resumed explicitly. Ada, however, provides an abort statement with which a task and all its dependent tasks can be abandoned. A restart of an aborted task is not possible.

3.3.3 Scheduling

A MASCOT Machine provides primitive operations for timing, synchronisation, and control of execution of Activities. The selection of an Activity for execution is controlled by a Scheduler.

3.3.3.1 Scheduler

An Activity is entered from the Scheduler and the period from control passing to the Activity until control passes back to the Scheduler is known as a Slice. Scheduling is defined in MASCOT in a manner which enables a set of algorithms to be designed for each implementation. These algorithms determine priority adjustments, Activity selection, Slice duration, and end of Slice action.

3.3.3.1.1 Adjustments to the Priority of an Activity

An Activity has a current priority which may influence Activity selection and Slice termination. The initial value of the priority is determined before the Activity commences its execution. It may be modified during processing.

Ada only allows static priorities of tasks using a special pragma. Thus, priorities cannot be altered during program execution.

3.3.3.1.2 The Selection of an Activity at the Start of a Slice

The scheduling algorithm selects an Activity which is ready to run according to the priority mechanisms.

A similar strategy is chosen by the scheduler provided by an Ada implementation.

The Ada Scheduler, however, also considers possible rendezvous.

3.3.3.1.3 The Duration of a Slice

A Slice may be ended by the Activity itself or as a result of actions following an event external to the Activity. A MASCOT Machine may adopt either co-operative scheduling (Slices are only ended by Activities themselves) or pre-emptive scheduling.

If an Ada program is not affected by interrupts, co-operative scheduling is performed. If interrupts are handled within an Ada program, they are served with the highest priority, and therefore pre-emptive scheduling is performed.

3.3.3.1.4 End of Slice Action

The scheduling algorithm determines, for each type of Slice termination, how the Scheduler treats Activities whose Slice has ended, since the reason for Slice termination may influence the Activity's entitlement to selection for a further Slice.

In Ada the reason for a Slice termination has no influence on the selection of the next action. The selection process only depends on the priorities of possible rendezvous and of tasks ready to run.

While the MASCOT definition does not define in detail any of these algorithms, an implementation of the MASCOT Machine precisely describes the algorithms chosen. Ada, however, prescribes the scheduling algorithms in the presence of priorities.

Except when caused directly by an interrupt, Slice termination results from an Activity calling one of the MASCOT primitive operations to provide:

- Synchronisation;

- Timing;

- Activity Suspension and Termination.

Slice terminations in an Ada environment result from:

- Entry Calls;

- Execution of Accept Statements;

- Execution of Delay Statements;

- Termination of Tasks;

- Interrupts.

3.3.3.2 Synchronisation

Synchronisation in MASCOT covers mutual exclusion of competing Activities, using the primitives JOIN and LEAVE, and cross-stimulation of co-operating Activities, using the primitives WAIT and STIM. Explicit synchronisation is achieved by the four primitive operations which operate on special objects called Control Queues. Since synchronisation takes place only in respect of access to IDAs, each Control Queue is conceptually part of an IDA's data structure. Synchronisation primitive operations are encapsulated within an access mechanism. The Control Queue is defined as an object on which the primitive operations have effects (refer [1]), and which may be given a priority specification to influence the scheduling algorithms.

Ada does not provide such primitive operations. Mutual exclusion is ensured if two tasks are in a rendezvous. Cross-stimulation is achieved by entry calls and accept statements. A Control Queue is not defined explicitly but it is implicitly built by an entry declaration. It is, however, possible to formulate the semantics of the MASCOT primitives in terms of Ada.

3.3.3.3 Timing

MASCOT provides for two timing primitives, DELAY and TIMENOW. An Activity may stop processing for at least a specified period of time by issuing the DELAY primitive which takes a single parameter expressing the delay period. An Ada task achieves the same effect by executing a delay statement.

The primitive, TIMENOW, returns the current absolute value of time in the same units as used for the delay parameter. An Ada implementation provides the function CLOCK in a predefined package CALENDAR which returns the current value of time.

3.3.3.4 Activity Suspension and Termination

The primitive, SUSPEND, is used by an Activity to return control to the Scheduler in order to achieve co-operative scheduling. The Activity stops processing until it is next scheduled. Ada does not provide such a primitive. The same effect, however, is achieved by executing a delay statement whose duration is zero.

The primitive, ENDROOT, is called immediately after the last executable statement of a Root so that the Activity defined by the Root can end correctly if and when execution of its code is complete. An Ada task does not need such a primitive. It terminates after execution of its last executable statement if all its dependent tasks have terminated.

3.3.4 Device Handling

A system interacts with the external world through a set of peripheral devices attached to the processors on which it is running. The primary functions of

these devices is to act as sources or sinks of data, providing input and receiving output. Normally an application system itself chooses when input-output operations are to be performed. A realtime application system, however, is driven by an external process and cannot in every case determine itself when input-output operations must be performed. Therefore the system must allow to be interrupted by external events and must serve those interrupts immediately.

The MASCOT Machine provides facilities to enable devices to be handled in the manner appropriate to the application. In [1] a model is described on which the facilities are based, with such modifications as the special features of particular implementations warrant.

Ada provides two facilities to communicate with the external world:

- an Input-Output Interface;

- the connection of task entries to interrupts.

Ada applications can easily adopt the MASCOT model of device handling.

3.3.5 Monitoring

MASCOT defines a mechanism for monitoring certain events in strict runtime order. The definition (refer [1]) specifies that events to be monitored may be selected, and how the monitoring is to be performed. A monitoring mechanism is not part of every kind of MASCOT Machine.

Ada does not define a mechanism for monitoring. However, the MASCOT monitoring mechanism can be adopted by an Ada environment in two different ways:

- (a) during the construction of an application system monitoring facilities may be implanted in the program code, and an interface provided for the user according to the MASCOT model;

- (b) an Ada Debug System may be constructed according to the MASCOT model.

The second alternative seems to be a better solution, because monitoring should only be used in the development phase of an application system.

3.3.6 Interacting

A Command Interpreter provides an operator with access to some of the MASCOT facilities. It is always provided in implementations of an Evolutionary MASCOT Machine but is optional in implementations of a Frozen Machine. The facilities accessible through the Command Interpreter may be those for construction, program control, and monitoring.

A Command Interpreter is also part of an Ada Programming Support Environment. There is no reason why the Command Interpreter of an APSE should not provide facilities similar to those of the Command Interpreter of a MASCOT Machine.

3.4 Activities

In a MASCOT orientated application system Activities are independent processes (tasks) which are interconnected via Inter-Communication Data Areas (IDAs). Activities are created from Root templates (refer 3.3.1.1.2).

A Root template does not know the structure of the data passed to or from an IDA using the particular access mechanism. However, to be able to write the body of a Root template a programmer must assume the type of the formal parameters of the particular access procedure.

A Root template written in terms of Ada can be a generic procedure. The value parameters then become the formal parameters of the procedure. The access procedures to the IDAs used are given as generic formal parameters. To be able to compile a Root template, i.e. to make it known to the "Construction Data Base", the visible data types of the IDAs needed must be imported using a context clause. Otherwise it will not compile without errors.

In terms of Ada a Root template comprises the following information:

- its name;

- a specification of the data types of those IDAs it will use, given in a context clause;

- a specification of access procedures, given as generic formal parameters;

- a specification of constant value parameters, given as formal procedure parameters of mode IN;

- its body.

Many Activities can be created from a Root template. The creation process is achieved by instantiating the generic procedure, which is the Root template, by supplying generic actual parameters. The generic actual parameters are the access procedures of the IDAs used by the particular Activity.

In terms of Ada, Activities are tasks. In a first approach such a task only contains the call of the instantiated "Root procedure". Tasks are neither compilation units nor library units and must therefore be encapsulated in subprograms or packages. This poses further constraints on forming an application system in Ada using MASCOT.

3.5 Inter-Communication Data Areas

MASCOT provides two kinds of Inter-Communication Data Areas (IDAs): Channels and Pools. The MASCOT Machine treats them as constructionally equivalent (refer 3.3.1.1.1).

Because of the constraints brought in by the formulation of Root templates in an Ada environment, an IDA must be split into two units: a definition of the data types it uses and a definition of its implementation, which makes the access mechanisms visible to Activities but hides the data area the IDA works on and the implementation of the access mechanisms. Control queues are not needed and several IDA templates can use the same definition of data types.

The data types of an IDA template are encapsulated in an Ada package. This approach allows the implementation of an abstract data type with clearly defined operations allowed on objects of this data type.

An IDA template itself is given as a generic package with no generic formal parameters. The specification of that package only consists of the access procedures visible to Roots. The data area an IDA is working on need not be visible to a Root which uses the IDA.

Many IDAs can be created from one IDA template by instantiating it. A disadvantage of the Ada approach is that any presets and initialisations of an IDA's data area must be part of the IDA template because of Ada rules.

Because Channels serve as unidirectional interfaces and only store temporarily the data items passed through them without processing the items, the data type of a Channel can be a generic formal parameter of a Channel template. Several different Channels can be created, which handle different data types, from a single template definition. An Ada environment should therefore distinguish between Channels and Pools.

The information in a Channel template comprises:

- its name;

- a specification of the data type it handles, given as a generic formal parameter;

- the access procedures which define the operations allowed on its data area, given in its visible part.

The information of a Pool template comprises:

- its name;

- a specification of the data types it handles, given as an imported package in its context clause;

- the access procedures, which define the operations allowed on its data area, given in its visible part.

Both templates hide the structure of their data areas, because this is not of any importance to the particular Activities, and the presets and initialisation procedures of their data areas.

3.6 Subsystems

Subsystems are formed out of Roots and IDAs (refer 3.3.1.3).

MASCOT knows two kinds of IDAs, if Subsystems are involved: Subsystem-IDAs and IDAs private to Subsystems. The private IDAs need not be visible outside of the particular Subsystem. Therefore the creation of private IDAs is performed during the formation of Subsystems. Subsystem-IDAs must be created prior to the forming process.

Activities are Ada tasks (which are not compilation units). Therefore the creation of Activities and the formation of Subprograms are performed together.

Activities are created directly from Root templates and not from Roots as required by MASCOT, because private IDAs, needed for the forming of Roots, are not visible outside Subsystems.

The forming process must be supplied with the following informations:

- the name of the Subsystem;

- the names of the packages which define the data types of the IDAs involved;

- the names of all already created Subsystem-IDAs;

- the names of all IDA templates to form its private IDAs;

- the names of all Root templates to create the Activities which the Subsystem consists of;

each Activity to be created must be supplied with the following information:

- its name;

- its actual value parameters;

- the names of the access procedures of the IDAs which it will use.

Root templates and IDA templates can be used more than once as can Subsystem-IDAs.

The formed Subsystem is a procedure in terms of Ada. Subsystems can also be supplied with value parameters.

3.7 Frozen and Evolutionary Systems

MASCOT knows two different implementations of the MASCOT Machine: a Frozen Machine and an Evolutionary Machine.

3.7.1 Frozen Systems

The implementation of a Frozen MASCOT Machine means that the application system is fully constructed before a start command is issued. No changes of the system are allowed during runtime. This is the normal non-MASCOT way of constructing an application system.

Using the MASCOT philosophy in combination with Ada the resulting application system can be a main program in Ada terminology. The main program must ensure that all Activities execute in parallel at a given time. Therefore it cannot call the procedures denoting Subsystems consecutively, because in this case every Subsystem would be forced to complete its execution before another Subsystem could start its execution. In the contrary the main program must ensure that all Subsystems execute in parallel. This means that the calls of the procedures denoting Subsystems must be encapsulated in tasks local to the main program.

Therefore a final construction stage is added to the construction stages defined by MASCOT: the formation of a main program from already formed Subsystems.

3.7.2 Evolutionary Systems

An implementation of a MASCOT Machine which allows the set of Subsystems constituting an application system to be changed during the execution of the application system by the addition and/or deletion of Subsystems is said to be Evolutionary. Such a MASCOT Machine is needed wherever an application system must not be interrupted while exchanging erroneous parts or adding new features. Consider, for example, an air traffic system. An Evolutionary System is also very useful during the development of any application system (Frozen or Evolutionary).

In an Evolutionary System Subsystems can be looked at as independent programs which communicate via Subsystem-IDAs and execute under control of a special runtime system which implements the related MASCOT features. The runtime system contains a dynamic linker/loader to be able to add and remove the executable code of Subsystems and link the Subsystems to the respective Subsystem-IDAs online. If needed, also the executable code of Subsystem-IDAs can be added or removed.

In Ada, Subsystems are procedures to which library packages implementing Subsystem-IDAs are visible. Ada procedures can be main programs in the usual sense. The means by which the execution of main programs is started is not defined by the Ada language. Therefore it is possible to implement an executive which operates like an Evolutionary MASCOT Machine and which treats Ada main programs as Subsystems. However, care must be taken concerning Subsystem-IDAs. Normally a Subsystem-IDA is used by more than one Subsystem. Therefore various Ada main programs (Subsystems) mention the same Subsystem-IDA in their context clauses. Thus, according to the language rules, the code of a Subsystem is elaborated as many times as it is mentioned in the context clause of a main program and possibly loaded more than once. The code of a Subsystem-IDA, however, should only exist once in the Evolutionary System. Therefore a special linker/loader is needed to ensure that a Subsystem is linked to all respective main programs (Subsystems) and only loaded and elaborated once.

Another solution, which treats the Subsystem-IDAs as main programs too, is not feasible in Ada, because program units denoting Subsystem-IDAs must be visible to the program units denoting Subsystems during compilation of the Subsystems to resolve and check the respective connectivity requirements.

An Ada programming support environment should facilitate version control of program units. Version control should include the ability of having different versions of the body of a program unit but of still using only one specification of the program unit. An Ada compiler only uses the specification of program units for checking connectivity constraints and establishing proper connections between program units (not always applicable for generic units and inline subprograms). This means that after the recompilation of a body its previously compiled version is not always to be made obsolete. A MASCOT Machine could utilize this property, although it would not offer the full range of facilities of an Evolutionary Machine, but it would allow to exchange the codes of Subsystems of a fixed set of Subsystems online. The solution needs

an Ada runtime system which allows to start, halt, resume, and terminate tasks interactively according to the respective MASCOT rules leaving the

overall Ada program in a consistent and correct state (an Ada debug system should offer such facilities anyway),

and

a dynamic linker/loader which allows to exchange bodies of program units online.

An application system is then formed like a Frozen System. The author feels that this approach is more appropriate concerning the Ada philosophy.

4. A Solution

The construction of Ada programs according to the MASCOT philosophy is possible. Some restrictions, however, have to be obeyed by programmers. The construction process is performed interactively, guided by the system. The system provides a set of formats from which System Element Templates and System Elements can be derived.

The proposed solution only deals with Frozen Systems and does not contain a monitoring system or a command interpreter. The requirements for the monitoring system and for the command interpreter can easily be derived from [1]. However, the proposed construction method can also be used for the construction of Evolutionary Systems, the formation of the Subsystems being the last stage then.

Eight construction stages are identified:

- (1) Creation of the ACP Diagram;
- (2) Definition of the types of those data objects, which will be handled by Inter-Communication Data Areas (IDAs), encapsulated in Ada packages;
- (3) Definition of the interfaces provided by the various IDA templates as specifications of generic Ada packages;
- (4) Definition of the Root templates as specifications of generic Ada procedures;
- (5) Implementation of the bodies of the Root and IDA templates;
- (6) Identification of Subsystem-IDAs and their instantiation;
- (7) Formation of the Subsystems;
- (8) Formation of the main program MASCOT_System.

If Subsystems are not considered, the stages (6) to (3) are performed as follows:

- (6) Instantion of the IDAs;
- (7) Formation of the main program MASCOT_System.

IDAs and Roots can be tested in test harnesses after implementation of the respective bodies.

4.1 The Construction Data Base

The definition of an ACP Diagram is the first stage of the construction of an application system using the MASCOT philosophy. The ACP Diagram defines the set of System Elements and the connections between them. However, an ACP Diagram only serves as an interface between a Construction Data Base and the development engineer.

The development engineer defines the ACP Diagram on a sheet of paper. An

interactive tool is used to provide this knowledge to the MASCOT development system. The tool constructs an internal graph of the ACP Diagram which is part of the Construction Data Base. The dialogue should be system driven to ensure the correctness and completeness of the design. Most parts of the construction process are automatic generation processes with only a few manual interactions. The final checks for correctness and completeness are performed by an Ada compiler.

4.1.1 Data Structure of the MASCOT Development System

Every possible object in a MASCOT development system has to be represented by an entry in the Construction Data Base. The following subchapters describe the different types of entries in terms of Ada.

4.1.1.1 Basic Types

Class types are used for parameterising several System Elements. System Elements and System Element Templates have names, called `name_string`, and their optional source codes are stored in files of name `file_name`.

```
TYPE system_class_type IS (with_subsystems , without_subsystems);
TYPE IDA_class_type IS (channel , pool);
SUBTYPE name_string IS string;
SUBTYPE file_name IS string;
```

4.1.1.2 Type Summary

4.1.1.2.1 Types of Entries

There must be a distinct type for every possible entry in the Construction Data Base. Lists of entries should be formable, too.

```
TYPE subsystem;
TYPE access_subsystem IS ACCESS subsystem;
TYPE subsystem_list_element;
TYPE access_subsystem_list_element IS ACCESS subsystem_list_element;

TYPE activity;
TYPE access_activity IS ACCESS activity;
TYPE activity_list_element;
TYPE access_activity_list_element IS ACCESS activity_list_element;
```

```

TYPE IDA(class : IDA_class_type);
TYPE access_IDA IS ACCESS IDA;
TYPE IDA_list_element;
TYPE access_IDA_list_element IS ACCESS IDA_list_element;

TYPE root_template;
TYPE access_root_template IS ACCESS root_template;
TYPE root_template_list_element;
TYPE access_root_template_list_element
    IS ACCESS root_template_list_element;

TYPE IDA_template(class : IDA_class_type);
TYPE access_IDA_template IS ACCESS IDA_template;
TYPE IDA_template_list_element;
TYPE access_IDA_template_list_element
    IS ACCESS IDA_template_list_element;

TYPE data_type;
TYPE access_data_type IS ACCESS data_type;
TYPE data_type_list_element;
TYPE access_data_type_list_element IS ACCESS data_type_list_element;

```

4.1.1.2.2 Parameter Types

Some of the entries are supplied with lists of parameters and of access procedures.

```

TYPE parameter;
TYPE access_parameter IS ACCESS parameter;
TYPE parameter_list_element;
TYPE access_parameter_list_element IS ACCESS parameter_list_element;

TYPE access_procedure;
TYPE access_access_procedure IS ACCESS access_procedure;
TYPE access_procedure_list_element;
TYPE access_access_procedure_list_element
    IS ACCESS access_procedure_list_element;

```

4.1.1.3 List Types

An object of a list type points to the list element it represents and to the next element in the list.

```

TYPE subsystem_list_element IS
    RECORD
        element : access_subsystem;
        next    : access_subsystem_list_element := NULL;
    END RECORD;

```

```

TYPE activity_list_element IS
  RECORD
    element : access_activity;
    next    : access_activity_list_element := NULL;
  END RECORD;

TYPE IDA_list_element IS
  RECORD
    element : access_IDA;
    next    : access_IDA_list_element := NULL;
  END RECORD;

TYPE root_template_list_element IS
  RECORD
    element : access_root_template;
    next    : access_root_template_list_element;
  END RECORD;

TYPE IDA_template_list_element IS
  RECORD
    element : access_IDA_template;
    next    : access_IDA_template_list_element;
  END RECORD;

TYPE data_type_list_element IS
  RECORD
    element : access_data_type;
    next    : access_data_type_list_element := NULL;
  END RECORD;

TYPE parameter_list_element IS
  RECORD
    element : access_parameter;
    next    : access_parameter_list_element := NULL;
  END RECORD;

TYPE access_procedure_list_element IS
  RECORD
    element : access_access_procedure;
    next    : access_access_procedure_list_element := NULL;
  END RECORD;

```

4.1.1.4 The Types of the Parameters

An object of either type "access_procedure" or "parameter" only contains a formal name and the associated actual name because the correct interrelations of other attributes (e.g. types, parameters of the access procedures) are checked by the Ada compiler. The automatic generation system only needs to know the names to be able to generate the proper format of the particular source code frame.

```

TYPE access_procedure IS
  RECORD
    formal : name_string;
    actual : name_string;
  END RECORD;

```

```

TYPE parameter IS
  RECORD
    formal : name_string;
    actual : name_string;
  END RECORD;

```

4.1.1.5 The Type "MASCOT_System"

An object of type "MASCOT_System" is the head of the description of the application system to be built. The system may or may not contain Subsystems. The usual name of the system should be "MASCOT_System".

If the system contains Subsystems, the object points to a list of Subsystems and Subsystem-IDAs. These lists are needed to build the correct context clause.

If the system does not contain Subsystems, the object points to a list of Activities and IDAs. The construction of the associated program is then performed in a similar way as the construction of a Subsystem.

```

TYPE MASCOT_System(class : system_class_type) IS
  RECORD
    name : name_string := "MASCOT_System";
    file : file_name;
    CASE class IS
      WHEN with_subsystems =>
        subsystems : access_subsystem_list_element;
        subsystem_IDAs : access_IDA_list_element;
      WHEN without_subsystems =>
        activities : access_activity_list_element;
        IDAs : access_IDA_list_element;
      END CASE;
  END RECORD;

```

```

TYPE access_MASCOT_System IS ACCESS MASCOT_System;

```

4.1.1.6 The Type "subsystem"

An object of type "subsystem" is the head of the description of a Subsystem. It contains a pointer to a list of

parameters describing the optional value parameters of the Subsystem together with their actual values;

Activities;

IDAs private to it;

Subsystem-IDAs.

The information is needed to build the proper context clause and to instantiate the Activities and the private IDAs.

TYPE subsystem IS

RECORD

name : name_string;
file : file_name;
parameters : access_parameter_list_element;
activities : access_activity_list_element;
IDAs : access_IDA_list_element;
subsystem_IDAs : access_IDA_list_element;
END RECORD;

4.1.1.7 The Type "activity"

An object of type "activity" describes an Activity as it is created within a Subsystem. The following attributes are needed:

- a pointer to the object which describes the Root template;
- a list of parameters which associate the formal value parameters of the Root template with the actual ones;
- a list of access procedures which associate the generic formal procedures of the Root template with the actual access procedures;
- a list of the accessed IDAs (This list is for information only).

TYPE activity IS

RECORD

name : name_string;
root : access_root_template;
parameters : access_parameter_list_element;
access_procedures : access_access_procedure_list_element;
IDAs : access_IDA_list_element;
END RECORD;

4.1.1.8 The Type "IDA"

An object of type "IDA" describes an IDA, whether Subsystem-IDA or IDA private to a Subsystem. The object must be constrained to denote whether it describes a Channel or a Pool. The component "file_name" contains an empty string, if the IDA is an IDA private to a Subsystem. The following attributes are needed:

- a pointer to the object which describes the IDA template;
- a list of the objects which denote the packages defining the data types

used by the IDA (normally this list only consists of one element);

if it is a Channel, a list of the generic actual parameters associated with the generic formal parameters of the template.

```
TYPE IDA(class : IDA_class_type) IS
  RECORD
    name      : name_string;
    file      : file_name;
    template  : access_IDA_template(class);
    data_types : access_data_type_list_element;
    CASE class IS
      WHEN pool    => NULL;
      WHEN channel => parameters : access_parameter_list_element;
    END CASE;
  END RECORD;
```

4.1.1.9 The Type "root_template"

An object of type "root_template" describes a Root template. The following attributes are needed:

the name of the file which contains the source code of the specification;

the name of the file which contains the source code of the body;

a list of its formal value parameters;

a list of generic formal procedures which denote the access procedures to IDAs;

a list of the objects which denote the packages defining the data types used by possible IDAs.

```
TYPE root_template IS
  RECORD
    name      : name_string;
    spec      : file_name;
    bodie     : file_name;
    parameters : access_parameter_list_element;
    access_procedures : access_access_procedure_list_element;
    data_types : access_data_type_list_element;
  END RECORD;
```

4.1.1.10 The Type "IDA_template"

An object of type "IDA_template" describes an IDA template. The object must be constrained to denote whether it is a Channel or a Pool. The following attributes are needed:

the name of the file which contains the source code of the specification;

the name of the file which contains the source code of the body;
 a list of the access procedures which are provided;
 if it is a Channel, a list of its generic formal parameters;
 if it is a Pool, a list of objects which denote the packages defining the data types used (normally this list consists of only one element).

```

TYPE IDA_template(class : IDA_class_type) IS
  RECORD
    name          : name_string;
    spec          : file_name;
    bodie         : file_name;
    access_procedures : access_access_procedure_list_element;
  CASE class IS
    WHEN pool      => data_types : access_data_type_list_element;
    WHEN channel   => parameters : access_parameter_list_element;
  END CASE;
END RECORD;

```

4.1.1.11 The Type "data_type"

An object of type "data_type" denotes the specification of types of those data objects handled by IDAs. If an abstract data type is not considered, the attribute "body" contains an empty string, because a body is not needed in this case.

```

TYPE data_type IS
  RECORD
    name : name_string;
    spec : file_name;
    bodie : file_name;
  END RECORD;

```

4.1.2 The ACP Diagram

An ACP Diagram is a directed graph. Activities and IDAs are the nodes, access procedures of the IDAs are the arcs. The arcs are directed, because some access procedures deliver data to IDAs, some deliver data from IDAs. Pools can have bidirectional access mechanisms, but it can be assumed that the main purpose of these access mechanisms is unidirectional. Consider an access to a pool, where the questioner provides a key to receive proper data, or vice versa.

The graph of an ACP Diagram describes the whole MASCOT_System. Subsystems are subgraphs. Subsystem-IDAs are owned by more than one subgraph.

The first step towards the MASCOT_System is the construction of a graph which denotes the ACP Diagram of the MASCOT_System. The node types are given by type "node_class_type", the arc types by type "arc_class_type". An arc always starts at a node of type "act" (activity). "From" means that the Activity receives

data from the connected IDA.

```
TYPE node_class_type IS (act , channel , pool);
TYPE arc_class_type IS (from , to);

TYPE node(class : node_class_type);
TYPE access_node IS ACCESS node;
TYPE node_list_element;
TYPE access_node_list_element IS ACCESS node_list_element;

TYPE arc;
TYPE access_arc IS ACCESS arc;
TYPE arc_list_element;
TYPE access_arc_list_element IS ACCESS arc_list_element;
```

The list of nodes forms the ACP Diagram. Every node is associated with a list of arcs. The implementation of the graph does not contain any information about Subsystems.

4.1.2.1 The Type "node"

An object of type "node" contains the following information:

- its name which is the name of the future System Element;
- a list of those arcs which start or end at the node;
- its class.

```
TYPE node IS
  RECORD
    name : name_string;
    arcs : access_arc_list_element;
    class : node_class_type;
  END RECORD;

TYPE node_list_element IS
  RECORD
    element : access_node;
    next : access_node_list_element := NULL;
  END RECORD;
```

4.1.2.2 The Type "arc"

An object of type "arc" contains the following information:

- its name which is the name of the future access procedure;
- its direction (this information is used to create a picture of the ACP Diagram automatically);

its source which is always an Activity;

its sink which is always an IDA.

```
TYPE arc IS
  RECORD
    name      : name_string;
    direction : arc_class_type;
    source     : access_node;
    sink       : access_node;
  END RECORD;

TYPE arc_list_element IS
  RECORD
    element : access_arc;
    next     : access_arc_list_element;
  END RECORD;
```

4.1.2.3 The Construction Procedure for the ACP Diagram

The ACP Diagram is constructed with the procedure "construct_ACP_Diagram". The ACP Diagram is delivered as a list of nodes (Activities and IDAs). The first phase of the construction process is the input of the various nodes. The second phase is the input of the arcs which are the access procedures of IDAs and are called by the Activities. Every node has a list of arcs. Therefore every arc is stored twice: at the node that is an Activity and at the node that is an IDA.

The procedure has the following interface:

```
WITH text_io, construction_data_base_types;
USE  text_io, construction_data_base_types;

PROCEDURE construct_ACP_Diagram(head : OUT access_node_list_element);
```

4.1.3 Creation of Data Types

Every IDA is connected with a package which defines the types of the data objects needed by the access procedures of the IDA. Various IDAs can share the same data type definitions. The Construction Data Base contains a list of those packages. The list is constructed with the procedure "form_data_types".

The interface of the procedure is:

```
WITH text_io, construction_data_base_types;
USE  text_io, construction_data_base_types;

PROCEDURE form_data_types
  (data_types_list : OUT access_data_type_list_element);
```

4.1.4 Creation of IDA Templates

A list of all IDA templates is created with the procedure "form_IDA_templates". Every template is supplied with a list of its access procedures. When inputting the access procedure names, care must be taken to ensure that they are the same as the names of the access procedures used in the ACP Diagram. Such a check cannot be performed by this procedure, because one template can be used to create several IDAs and because this procedure has no access to the ACP Diagram.

In the case of a Pool template the list object is supplied with the proper element which describes the data types used by its access procedures. In the case of a Channel template the list object is supplied with formal data types (and operations on them).

The interface of the procedure is:

```
WITH text_io, construction_data_base_types;
USE  text_io, construction_data_base_types;

PROCEDURE form_IDA_templates
  (data_types_list  : access_data_type_list_element;
   IDA_template_list : OUT access_IDA_template_list_element);
```

4.1.5 Creation of Root Templates

A list of Root templates is created with the procedure "form_root_templates". Every template is supplied with a list of

the packages which define the data types used by the various access procedures to IDAs;

formal access procedures to IDAs;

formal value parameters of the Root template.

The interface of the procedure is:

```
WITH text_io, construction_data_base_types;
USE  text_io, construction_data_base_types;

PROCEDURE form_root_templates
  (data_types_list  : access_data_type_list_element;
   root_template_list : OUT access_root_template_list_element);
```

4.1.6 Forming of Subsystems

Subsystems are formed with the procedure "form_Subsystems". If no Subsystems are considered, the procedure delivers an empty list of Subsystems.

If Subsystems are considered, the forming process consists of four phases:

(1) Construction of the Subsystems

A Subsystem is constructed by naming the Activities which it consists of. A check is made to ensure that an Activity is only used once. The IDAs which belong to a Subsystem can then be derived from the ACP Diagram automatically. Every Subsystem is supplied with a list of formal value parameters.

(2) Forming of the Activities

Activities are created from Root templates by supplying the template with the actual value parameters and with the actual access procedures to IDAs. A check is made to ensure that the actual access procedures are equivalent to those mentioned in the ACP Diagram.

(3) Forming of the Subsystem-IDAs and the IDAs private to Subsystems

IDAs are created from IDA templates. The system determines whether an IDA is private to a Subsystem or whether it is a Subsystem-IDA. If the template is a Channel template, the IDA is supplied with the name of the proper data type definition and with the actual data types. A check is made to ensure that the class of the IDA given in the ACP Diagram and the class of the IDA template match.

(4) Supplying every Activity with a list of those IDAs connected to it

Every Activity is supplied with a list of those IDAs to which it has access.

The interface of the procedure is:

```
WITH text_io, construction_data_base_types;
USE text_io, construction_data_base_types;

PROCEDURE form_Subsystems
  (data_types_list : access_data_type_list_element;
   IDA_template_list : access_IDA_template_list_element;
   root_template_list : access_root_template_list_element;
   ACP_Diagram : access_node_list_element;
   subsystem_list : OUT access_subsystem_list_element);
```

4.1.7 Forming of the Main Program

The main program is formed with the procedure "form_MASCOT_System". The procedure consists of two independent parts.

If Subsystems are considered (the list of Subsystems is not empty), every Subsystem is supplied with its actual value parameters and a list of all Subsystem-IDAs is constructed.

If Subsystems are not considered, Activities and IDAs are formed according to the ACP Diagram. The forming process is similar to that performed during forming of Subsystems.

The interface of the procedure is:

```
WITH text_io, construction_data_base_types;  
USE text_io, construction_data_base_types;
```

```
PROCEDURE form_MASCOT_System  
  (data_types_list      : access_data_type_list_element;  
   IDA_template_list    : access_IDA_template_list_element;  
   root_template_list   : access_root_template_list_element;  
   subsystem_list       : IN OUT access_subsystem_list_element;  
   ACP_Diagram           : access_node_list_element;  
   MASCOT_System_object : OUT access_MASCOT_System);
```

4.1.8 The Main Program of the Construction Process

The main program of the construction process of the Construction Data Base calls successively the several subprograms which implement the various construction stages. It only implements the first generation of an application program using the MASCOT philosophy. The programs with which this generation can be changed are not described in this paper.

The subprograms "generate_..._frame" then build the frames of all System Elements and System Element Templates. The implementations of the subprograms are not described in this report, because machine dependent file accesses have to be performed. However, an implementation can easily be derived from the frames given in the following subchapters.

The Construction Data Base is held in main memory during the construction process. The subprograms "save_MASCOT_System" and "save_ACP_Diagram" store the whole Data Base on mass storage to allow further work on the Data Base. These subprograms are heavily machine dependent and their implementation is therefore not given in this report.


```

WITH construction_data_base_types;
WITH construct_ACP_Diagram;
WITH form_MASCOT_System;
WITH form_Subsystems;
WITH form_data_types;
WITH form_IDA_templates;
WITH form_root_templates;
WITH generate_data_types_frame, generate_root_templates_frame,
    generate_IDA_templates_frame, generate_subsystem_frame,
    generate_main_program_frame;
WITH save_MASCOT_System;
WITH save_ACP_Diagram;
USE construction_data_base_types;

PROCEDURE construct_data_base
IS
    ACP_Diagram          : access_node_list_element;
    MASCOT_System_object : access_MASCOT_System;
    data_types_list      : access_data_type_list_element;
    IDA_template_list    : access_IDA_template_list_element;
    root_template_list   : access_root_template_list_element;
    subsystem_list       : access_subsystem_list_element;
BEGIN
    construct_ACP_Diagram(ACP_Diagram);

    form_data_types(data_types_list);
    generate_data_types_frame(data_types_list);

    form_IDA_templates(data_types_list , IDA_template_list);
    generate_IDA_templates_frame(IDA_template_list);

    form_root_templates(data_types_list , root_template_list);
    generate_root_templates_frame(root_template_list);

    form_Subsystems(data_types_list ,
                    IDA_template_list ,
                    root_template_list ,
                    ACP_Diagram ,
                    subsystem_list);
    generate_subsystem_frame(subsystem_list);

    form_MASCOT_System(data_types_list ,
                    IDA_template_list ,
                    root_template_list ,
                    subsystem_list ,
                    ACP_Diagram ,
                    MASCOT_System_object);
    generate_main_program_frame(MASCOT_System_object);

    save_MASCOT_System(MASCOT_System_object , data_types_list ,
                    IDA_template_list , root_template_list);
    save_ACP_Diagram(ACP_Diagram);
END construct_data_base;

```

4.2 Data Types of the IDAs

Activities communicate with each other via IDAs. An IDA provides a procedural interface for that purpose. Because IDAs store data passed through them either permanently (in the case of Pools) or temporarily (in the case of Channels) in a private data area, the procedural interface of an IDA is a set of access procedures to this data area.

Activities are created from Root templates. A Root template need not know with which IDA a derived Activity will be connected. The Root template, however, must specify access procedures to a set of possible IDAs in a formal manner because of Ada rules. The types of the data objects passed with these formal procedures must also be known by the Root template. An Activity derived from a Root template can then be only connected with such IDAs whose access procedures can be matched with the formal procedures of the Root template.

IDAs are created from IDA templates. IDA templates already provide the procedural interface of the future IDA. The template also implements the data area of the future IDA. Therefore the types of the data objects passed through the future IDA must be already known by the IDA template.

```
PACKAGE <name of data_types> IS
    -- declaration of the data types (or abstract data type)
    -- used by IDAs
END <name of data_types>;
```

Figure 4.2-1: Frame of a Package containing data types
for IDAs

The definition of the particular data types must therefore be visible to IDA template as well as to Root template. To ease the construction of the application system the definition of the data types of a single IDA should therefore be subsumed under a single package specification in terms of Ada. An IDA template then names exactly one definition of data types in its context clause, a Root template names as many as there are connections with particular IDAs for a derivable Activity. It may be that the types of objects passed through different IDAs are equivalent. Therefore different IDA templates can share the same definition of data types.

An Ada package can define an abstract data type which consists of any number of type definitions and a set of allowed operations on the types. Such abstract data types can easily be handled if the MASCOT philosophy is used. In this case a package body is needed additionally to the package which defines the data types of an IDA.

4.3 The Form of IDA Templates

Ada provides the facility to specify templates of program units with its generic concept. Procedural interfaces, as needed by an IDA template, can only be formed by using packages. Therefore an IDA template will be a generic package in an Ada environment.

4.3.1 Channel Template

In Ada it is possible to restrict the use of objects of particular types. Such types are called (limited) private types and can only be declared in package specifications. The only operations available for objects of private types outside the declaring package are assignment, test for equality, and operations which are expressed as procedures/functions and which are declared in the same package specification. The assignment operation and the test for equality are not available for objects of limited private types. Generic formal types can be declared as private. Only assignment, test for equality, and associated generic formal subprograms are available for objects of those formal types within the generic unit.

```
-----  
GENERIC
```

```
    TYPE channel_data IS PRIVATE; -- type of those data objects  
                                   -- passed through the Channel;  
                                   -- perhaps more generic formal  
                                   -- parameters are needed to provide  
                                   -- operations on the data objects
```

```
PACKAGE <name of Channel template> IS
```

```
    -- declaration of the access_procedures
```

```
END <name of Channel template>; -- specification  
-----
```

Figure 4.3-1: Frame of the Specification of a Channel Template

A MASCOT Channel does not process data passed through it. To ensure mutual exclusion of competing Activities and to allow the delivery of more than one object to be passed before an object is consumed by an Activity, data objects must be stored temporarily in a private data area. The characteristics of the type of an object is not of any interest to the Channel. To allow temporary storage, assignment must be available for objects within the Channel. Therefore the type of these objects can be private (hidden from the Channel).

A template which denotes a Channel therefore does not need to be associated with a package which defines the data types of the future IDA but must have a generic formal parameter which denotes the type of the object passed through the

Channel. Perhaps more than one generic formal parameter is needed for that purpose.

Figure 4.3-1 shows the format of the specification of a Channel template.

4.3.2 Pool Template

According to the MASCOT philosophy a Pool is intended to hold data for a long period of time. A Pool processes the data objects more or less to be able to store or retrieve them in an efficient manner.

```
-----  
WITH <name of package that defines the data type>;  
USE  <name of package that defines the data type>;  
  
GENERIC  
  
PACKAGE <name of Pool template> IS  
    -- declaration of the access procedures  
  
END <name of Pool template>; -- specification  
-----  
  
Figure 4.3-2: Frame of the Specification of  
a Pool Template  
-----
```

The procedures, which implement a Pool in an Ada environment, must be allowed to perform all possible operations on data objects delivered to them. Therefore the types of the data objects must not be hidden from a Pool. Because a template already implements the behaviour of the future Pool, the types of the data objects must be made visible to the template.

In Ada a Pool template is a generic package which has no generic formal parameters but which mentions the package defining the data types of the future Pool in its context clause.

Figure 4.3-2 shows the format of the specification of a Pool template.

4.4 The Form of Root Templates

Communication between Activities is only allowed via IDAs. The actual communication links are built up during the creation of Activities. The particular IDAs are named during this process.

An Activity is an active part of an applicaiton system, whereas an IDA is a passive part, because an IDA only becomes active, if one of its access

procedures is invoked by an Activity. Therefore the template of an Activity is a generic procedure in an Ada environment (There are no generic tasks in Ada).

```
WITH <list of data types>; -- all packages declaring the data types
                           -- of IDAs used by the root must be
                           -- mentioned here
USE <list of data types>;

GENERIC

    -- declaration of the formal procedural interface to IDAs

PROCEDURE <name of Root template> -- (formal parameters, if any)
    ;
```

Figure 4.4-1: Frame of the Specification of a Root Template

Because of Ada rules the access procedures to possible IDAs must be known by the Root template. To allow a certain amount of parameterisation those procedures are generic formal parameters of the template.

The type of data objects which are passed by the access procedures should also be parameterisable but this would restrict the use of objects derived from those types because of Ada rules. A Root, however, creates and processes those data objects. Therefore their types cannot be private (hidden from the Root template) as they should be to allow all kinds of types as actual parameters for the generic formal types, but they must be fully known. This means that the definitions of the data types of all accessible IDAs must be made visible to the Root template.

Unfortunately this approach restricts the derivation of Activities from the Root template, because an Activity can then only communicate via IDAs whose access procedures match the generic formal procedures of the particular Root template.

The format of the specification of a Root template is shown by Figure 4.4-1.

4.5 Implementation of Roots and IDAs

Access procedures and data types form the interfaces of Activities and IDAs to their particular surroundings. Their implementations are hidden from each other. However, some restrictions are to be obeyed.

4.5.1 Roots

All features of Ada can be used for the implementation of a Root template. Communication with other parts of the system should only be made, however, by using the formal access procedures to IDAs. Tasks also should not be used within a Root template because according to the MASCOT philosophy the only tasks are Activities which are derived from Root templates.

```
-----  
PROCEDURE <name of Root template> -- (formal parameters, if any)  
IS  
  
    -- declaration of local objects  
  
BEGIN  
  
    -- body of the Root template  
  
END <name of Root template>; -- body  
-----
```

Figure 4.5-1: Frame of the Body of a Root Template

The value parameters of a Root template can be used, for example, to select special strategies in accessing IDAs.

4.5.2 IDAs

IDAs are the interfaces between Activities. Activities are executed in parallel. The implementation of an IDA must ensure that no conflicts arise between Activities while accessing the data area. Access to a data area therefore must be embedded in critical regions. Ada offers the rendezvous mechanism for that purpose.

MASCOT provides the four primitives JOIN, LEAVE, WAIT, and STIM for the implementation of critical regions. The primitives operate on special objects called control queues. If an Activity wants to talk to an IDA, it performs a JOIN operation on the associated control queue. After successful communication it LEAVES the control queue. The primitive JOIN indicates the entrance into a critical region, the primitive LEAVE the end of a critical region. The primitives STIM and WAIT are used for cross stimulation. If an Activity has successfully written into the data area of an IDA it performs a STIM operation on the associated control queue to indicate that data objects are available in the data area. If an Activity wants to read data from the data area of an IDA it performs a WAIT operation to test whether there are data objects available in the data area.

The implementer of Activities (or Root templates) should not be concerned with those primitives. Therefore the calls of the primitives are embedded in the

access procedures of IDAs. The MASCOT philosophy in an Ada environment should facilitate the same approach. The package which implements an IDA therefore only provides a procedural interface. The implementations of the procedures must ensure mutual exclusion of Activities which want to access the data area of an IDA. On the other hand, Ada procedures are reentrant. Therefore it is possible, for example, to calculate access strategies before entering a particular critical region, or, even more, to decide that mutual exclusion is not necessary (for example, if a read only access is considered).

```

PACKAGE BODY <name of Channel/Pool template> IS

    -- declaration of the data area that either temporarily
    -- holds data items sent through the Channel of stores
    -- data items hold in the Pool

    -- declaration of the task that implements the Channel/Pool

    -- declaration of the bodies of the access_procedures

BEGIN

    -- initialisation of the Channel/Pool

END <name of Channel/Pool template>; -- body

```

Figure 4.5-2: Frame of the Body of a Channel/Pool Template

Control queues are kinds of semaphores. The JOIN and LEAVE primitives are the P and V primitives respectively. The only possible implementation of a semaphore is its implementation as a task with two entries in an Ada environment. Therefore, if JOIN and LEAVE are to be implemented, the respective control queue must be expressed as a task.

```

TASK control_queue IS
    ENTRY join;
    ENTRY leave;
END control_queue;

TASK BODY control_queue IS
BEGIN
    LOOP
        ACCEPT join;
        ACCEPT leave;
    END LOOP;
END control_queue;

```

Such a task is needed for every control queue in the application system. This would result in heavy context switching and would slow down the system performance.

The primitives WAIT and STIM must be implemented in a similar manner. The

solution, however, is more complicated in their case. Therefore another approach should be considered.

If a task calls an entry and the called task is not able to accept the entry call immediately, the calling task is blocked and has to wait until the call is accepted. The primitives JOIN and LEAVE therefore become obsolete in an Ada environment. On the other hand an accept statement contains an optional sequence of statements for whose execution mutual exclusion is ensured. Therefore the access procedures of an IDA should be transformed into entry calls of a task. This task then handles all operations on the data area of an IDA. An Activity, however, does not know that its accesses to IDAs are treated in this way.

Consider the following example of a Channel which only passes one data object at a time. The task which implements the behaviour of the Channel then looks like as follows:

```
TASK channel IS
  ENTRY put(x : IN item);
  ENTRY get(x : OUT item);
END channel;

TASK BODY channel IS
  store_x : item;
BEGIN
  LOOP
    ACCEPT put(x : IN item) DO
      store_x := x;
    END;
    ACCEPT get(x : OUT item) DO
      x := store_x;
    END;
  END LOOP;
END channel;
```

Although this example is very simple, very complicated strategies for accessing an IDA's data area are implementable. The reader is referred to [3] for a more detailed discussion. Alternative solutions are also considered in [3], but the solution shown in this paper is the one more suited to Ada.

The data area of an IDA is built from the types defined in a special package for every IDA. Initialisation of the data area must be performed by the IDA itself. Therefore initialisation routines must be implemented within the package body of the IDA template.

4.6 Creation of IDAs

The creation of an IDA is achieved by instantiating the package, which denotes the particular IDA template. Generic actual parameters must be provided for the instantiation process. A Pool template does not have generic formal parameters therefore no actual ones must be supplied. In the case of a Channel the actual data types must be made visible and the generic formal parameters must be matched with the actual ones. Figure 4.6-1 shows the two kinds of instantiations.


```

WITH <name of package that defines the data type>;
USE <name of package that defines the data type>;

PACKAGE <name of Channel> IS
  NEW <name of Channel template> (channel_data => <actual type>);

```

```

PACKAGE <name of Pool> IS
  NEW <name of Pool template>;

```

Figure 4.6-1: Frames of Subsystem-IDAs

Ada instantiations can be library units as well as the generic units they are derived from. Subsystem-IDAs must be addressable by several Subsystems, because they do not belong to any one Subsystem. In Ada they have to be library units and are instantiated as such. But note: If two Subsystem-IDAs with exactly the same behaviour and data types are needed in an application system, the respective template must be instantiated twice. One instantiation creates exactly one library unit. Mentioning them in various context clauses does not multiply their codes.

IDAs private to a Subsystem need not to be visible outside the Subsystem. Therefore they are instantiated within the code of a Subsystem.

4.7 The Form of Subsystems

Closely related Activities and the respective IDAs are subsumed under a Subsystem. Subsystems communicate through those IDAs (Subsystem-IDAs) which form the communication links between Activities belonging to different Subsystems. They are not part of a Subsystem.

A Subsystem is a procedure in an Ada environment, because it is directly invoked by the main program and it has not to provide a large interface to its surroundings except to allow it to be called. A Subsystem has an optional list of formal value parameters. Value parameters can be used, for example, to select special strategies within a Subsystem or to supply the encapsulated Activities with actual value parameters.

Subsystem-IDAs are library units. A Subsystem which communicates via a particular Subsystem-IDA must import this Subsystem-IDA in its context clause.

The IDAs private to a Subsystem are instantiated in the same manner as Subsystem-IDAs within the Subsystem. The packages defining the data types of these IDAs and the respective IDA templates must be imported by the Subsystem in its context clause.

```

WITH <list of the data type specifications of the private IDAs>;
WITH <list of templates of those IDAs private to the Subsystem>;
WITH <list of to be used Root templates>;
WITH <list of the Subsystem-IDAs>;
USE <list of the data type specifications of the private IDAs>;
USE <list of templates of those IDAs private to the Subsystem>;
USE <list of to be used Root templates>;
USE <list of the Subsystem-IDAs>;

PROCEDURE <name of Subsystem> -- (formal parameters, if any)
IS
    -- instantiate private IDAs according to the following schema:
    --
    ---- Channel:
    ----
    ----     PACKAGE <name of Channel> IS
    ----         NEW <name of Channel template> (channel_data
    ----                                     => <actual type>);
    ----
    ---- Pool:
    ----
    ----     PACKAGE <name of Pool> IS
    ----         NEW <name of Pool template>;

    -- form the Activities according to the following schema:
    --
    ---- TASK <name of Activity>;
    ----
    ---- TASK BODY <name of Activity> IS
    ----
    ----     instantiate proper Root according to following schema:
    ----
    ----     PROCEDURE <name of Activity>_Root IS
    ----         NEW <name of Root template> (<formal parameters>
    ----                                     => <actual parameters>);
    ----
    ---- BEGIN
    ----
    ----     <name of Activity>_Root -- (actual parameters, if any)
    ----                             ;
    ----
    ---- END <name of Activity>; -- body

BEGIN

    NULL; -- may be replaced by monitor operations

END <name of Subsystem>;

```

Figure 4.7-1: Frame of a Subsystem

Activities are executing in parallel competing for access to IDAs. Ada offers a tasking concept to express parallelism. Tasks cannot be library units. Therefore tasks denoting Activities are embedded into Subsystems. A Subsystem declares as many different tasks as it contains Activities.

According to MASCOT Activities are created from Roots and Roots from Root templates. These two steps are performed in one in an Ada environment. A proper Root template is instantiated within the task denoting the particular Activity. This instantiation is similar to the creation of a Root in a MASCOT environment. The instantiation process is supplied with the proper actual access procedures to the respective IDAs to replace the generic formal ones of the Root template. The Activity-task then simply calls the instantiated Root-procedure by supplying actual value parameters.

An alternative could be to instantiate Root templates outside a Subsystem to be able to use the same instance of a Root template twice and to be fully in accordance with the MASCOT philosophy. This seems to be possible because procedures are reentrant in Ada. The first disadvantage, however, is that local variables of an instance then only exist once which can lead to strange effects. The second one is that all IDAs must also be instantiated outside the Subsystems.

A Subsystem must mention all Root templates which will be used in its context clause.

The declared tasks denoting Activities are initiated in parallel before the Subsystem starts the execution of its own list of statements. Normally this list only contains a NULL statement, because the Subsystem must await the completion of the Activity-tasks. However, the list of statements can consist of monitor operations. In this case Root as well as IDA templates must offer respective interfaces.

The overall format of a Subsystem is shown by Figure 4.7-1.

4.8 The Form of the Main Program with Subsystems

Subsystems execute in parallel at the system level. Therefore they are embedded in Ada tasks. The main program declares as many tasks as Subsystems exist. The body of such a task comprises only one statement: the call of the procedure which denotes the particular Subsystem. Every procedure call is supplied with actual parameters for the value parameters of the particular Subsystem. The main program mentions all Subsystems in its context clause.

The main program does not need to execute statements itself because it must await the completion of the Subsystem-tasks anyway. Its statement list therefore only comprises the NULL statement. On the other hand monitor operations can be performed easily by the main program instead of executing a NULL statement. This case, however, must be considered by the implementations of the various parts of the system.

The forming of the main program is the last step in creating an application system according to the MASCOT philosophy in an Ada environment. After this step the whole system must have been compiled. The compilation is the last check for correctness of the program, especially of the interfaces. The first check is performed during the building of the Construction Data Base.

Ada defines a strict compilation order. Therefore the various parts of the system have to be compiled in the following order: packages which define the data types of IDAs -> IDA templates -> Root templates -> instantiations of those IDA templates from which Subsystem-IDAs are derived -> Subsystems -> main program. However, every part can be compiled after its completed coding as long as the correct compilation order is ensured.

```

WITH <list of all Subsystems>;

PROCEDURE MASCOT_System
IS
    -- declare all Subsystems as tasks according to the following schema:
    --
    ---- TASK <name of Subsystem>_task;
    ----
    ---- TASK BODY <name of Subsystem>_task IS
    ---- BEGIN
    ----     <name of Subsystem> -- (actual parameters, if any)
    ----     ;
    ----
    ---- END <name of Subsystem>_task;

BEGIN

    NULL; -- may be replaced by monitor operations

END MASCOT_System;

```

Figure 4.8-1: Frame of the Main Program Consisting of Subsystems

4.9 The Form of the Main Program without Subsystems

Forming a main program without Subsystems is just like forming a Subsystem despite the fact that Subsystem-IDAs are not concerned (refer Subchapter 4.7).

The compilation order is: packages which define the data types of IDAs -> IDA templates -> Root templates -> main program.

```

WITH <list of data type specifications for IDAs>;
WITH <list of IDA templates>;
WITH <list of Root templates>;
USE <list of data type specifications for IDAs>;
USE <list of IDA templates>;
USE <list of Root templates>;

PROCEDURE MASCOT_System
IS
    -- instantiate IDAs according to the following schema:
    --
    ---- Channel:
    ----     PACKAGE <name of Channel> IS
    ----         NEW <name of Channel template> (channel_data
    ----                                     => <actual type>);
    ----
    ---- Pool:
    ----
    ----     PACKAGE <name of Pool> IS
    ----         NEW <name of Pool template>;

    -- form the Activities according to the following schema:
    --
    ---- TASK <name of Activity>;
    ----
    ---- TASK BODY <name of Activity> IS
    ----
    ----     instantiate proper Root according to the following schema:
    ----
    ----     PROCEDURE <name of Activity>_Root IS
    ----         NEW <name of Root template> (<formal parameters>
    ----                                     => <actual parameters>);
    ----
    ---- BEGIN
    ----
    ----     <name of Activity>_Root -- (actual parameters, if any)
    ----                             ;
    ----
    ---- END <name of Activity>;

BEGIN

    NULL; -- may be replaced by monitor operations

END MASCOT_System;

```

Figure 4.9-1: Frame of the Main Program without Subsystems

5. Final Remarks

MASCOT is a good methodology for the decomposition of large systems into smaller parts. It enables systems designers to handle a system in an easier way. However, MASCOT introduces an own view on parallelism within an application program. This complicates the use of the MASCOT philosophy in combination with a programming language which offers tasking facilities.

This paper has combined the programming language Ada and the MASCOT philosophy. It has found that the MASCOT philosophy is applicable to the development of Ada programs. However, the communication mechanisms of MASCOT are based on very basic synchronisation mechanisms using binary semaphores. Ada offers a more advanced solution with its rendezvous concept. A rendezvous actually implements a critical region. Therefore MASCOT's synchronisation primitives are already embedded in Ada's rendezvous. Their explicit use in an Ada program is neither recommended nor reasonable, because it would result in a very strange programming style.

MASCOT introduces a kind of controlled separate compilation including the check for correctness of interfaces with its template and construction data base concepts. Ada offers these facilities with its separate compilation concept and its generic concept. The construction of test beds and the simulation of the behaviour of parts of an application system are also easily possible in an Ada environment. However, Ada's views are slightly different, and more checks are performed, for example, type checking. Therefore the unchanged mapping of MASCOT's view should not be done because some power of Ada may be lost then.

A MASCOT system is composed of Activities and interfaces (IDAs) between them. An Ada program is a set of procedures which are successively called by a main program (procedure). Some procedures can contain parallel activities to quicken computation.

Mapping MASCOT onto Ada totally changes the Ada way of constructing a system and writing Ada programs becomes unnecessarily complicated. It is therefore recommended not to use the entire MASCOT methodology for the development of Ada programs, even though the author believes it to be possible. However, some concepts of MASCOT (for example: the use of clearly defined interfaces between tasks whose implementation is hidden from the tasks, and the monitoring concept) are very reasonable and should be taken as requirements for the development of an Ada programming support environment.

6. References

- [1] MASCOT Suppliers Association
The Official Handbook of MASCOT
5 December 1980
- [2] Reference Manual for the Ada Programming Language
ANSI/MIL-STD 1815 A
January 1983
- [3] C1719 - Further Ada Studies
Use of MASCOT in the Mapse
Document No: C1719/REP/10 issue 1
August 1982

APPENDICES

Appendix 1 lists the codes of the Ada programs which are used for the initial building of the Construction Data Base which describes an application system. The contents of Appendix 1 are:

construction_data_base_types	A1-1
construct_ACP_Diagram	A1-6
find_data_type	A1-9
find_node	A1-10
find_root	A1-11
find_IDA_template	A1-12
match_parameters	A1-13
match_procedures	A1-14
form_data_types	A1-16
form_IDA_templates	A1-17
form_root_templates	A1-20
form_Subsystems	A1-23
form_MASCOT_System	A1-33
construct_data_base	A1-39

Appendix 2 describes a simple example. The contents of Appendix 2 are:

1. The ACP Diagram	A2-1
2. The Construction Data Base	A2-3
2.1 The Example with Subsystems	A2-3
2.2 The Example without Subsystems	A2-11
3. Data Types of the IDAs	A2-12
4. The IDA Templates	A2-12
4.1 IDA Template 1	A2-12
4.2 IDA Template 2	A2-13
4.3 IDA Template 3	A2-15
5. The Root Templates	A2-16
5.1 Root Template 1	A2-16
5.2 Root Template 2	A2-17
5.3 Root Template 3	A2-17
6. Subsystem-IDAs	A2-18
7. Subsystems	A2-18
8. The Main Program with Subsystems	A2-20
9. The Main Program without Subsystems	A2-21

PACKAGE construction_data_base_types
IS

```
TYPE system_class_type IS (with_subsystems , without_subsystems);
TYPE IDA_class_type IS (channel , pool);
SUBTYPE name_string IS string;
SUBTYPE file_name IS string;

TYPE subsystem;
TYPE access_subsystem IS ACCESS subsystem;
TYPE subsystem_list_element;
TYPE access_subsystem_list_element IS ACCESS subsystem_list_element;

TYPE activity;
TYPE access_activity IS ACCESS activity;
TYPE activity_list_element;
TYPE access_activity_list_element IS ACCESS activity_list_element;

TYPE IDA(class : IDA_class_type);
TYPE access_IDA IS ACCESS IDA;
TYPE IDA_list_element;
TYPE access_IDA_list_element IS ACCESS IDA_list_element;

TYPE root_template;
TYPE access_root_template IS ACCESS root_template;
TYPE root_template_list_element;
TYPE access_root_template_list_element IS
    ACCESS root_template_list_element;

TYPE IDA_template(class : IDA_class_type);
TYPE access_IDA_template IS ACCESS IDA_template;
TYPE IDA_template_list_element;
TYPE access_IDA_template_list_element IS ACCESS IDA_template_list_element;

TYPE data_type;
TYPE access_data_type IS ACCESS data_type;
TYPE data_type_list_element;
TYPE access_data_type_list_element IS ACCESS data_type_list_element;

TYPE parameter;
TYPE access_parameter IS ACCESS parameter;
TYPE parameter_list_element;
TYPE access_parameter_list_element IS ACCESS parameter_list_element;

TYPE access_procedure;
TYPE access_access_procedure IS ACCESS access_procedure;
TYPE access_procedure_list_element;
TYPE access_access_procedure_list_element IS
    ACCESS access_procedure_list_element;
```

```

TYPE subsystem_list_element IS
  RECORD
    element : access_subsystem;
    next    : access_subsystem_list_element := NULL;
  END RECORD;

TYPE activity_list_element IS
  RECORD
    element : access_activity;
    next    : access_activity_list_element := NULL;
  END RECORD;

TYPE IDA_list_element IS
  RECORD
    element : access_IDA;
    next    : access_IDA_list_element := NULL;
  END RECORD;

TYPE root_template_list_element IS
  RECORD
    element : access_root_template;
    next    : access_root_template_list_element;
  END RECORD;

TYPE IDA_template_list_element IS
  RECORD
    element : access_IDA_template;
    next    : access_IDA_template_list_element;
  END RECORD;

TYPE data_type_list_element IS
  RECORD
    element : access_data_type;
    next    : access_data_type_list_element := NULL;
  END RECORD;

TYPE parameter_list_element IS
  RECORD
    element : access_parameter;
    next    : access_parameter_list_element := NULL;
  END RECORD;

TYPE access_procedure_list_element IS
  RECORD
    element : access_access_procedure;
    next    : access_access_procedure_list_element := NULL;
  END RECORD;

```

```

TYPE access_procedure IS
  RECORD
    formal : name_string;
    actual : name_string;
  END RECORD;

TYPE parameter IS
  RECORD
    formal : name_string;
    actual : name_string;
  END RECORD;

TYPE MASCOT_System(class : system_class_type) IS
  RECORD
    name : name_string := "MASCOT_System";
    file : file_name;
    CASE class IS
      WHEN with_subsystems =>
        subsystems      : access_subsystem_list_element;
        subsystem_IDAs : access_IDA_list_element;
      WHEN without_subsystems =>
        activities : access_activity_list_element;
        IDAs       : access_IDA_list_element;
      END CASE;
    END RECORD;

TYPE access_MASCOT_System IS ACCESS MASCOT_System;

```

TYPE subsystem IS

RECORD

 name : name_string;
 file : file_name;
 parameters : access_parameter_list_element;
 activities : access_activity_list_element;
 IDAs : access_IDA_list_element;
 subsystem_IDAs : access_IDA_list_element;
END RECORD;

TYPE activity IS

RECORD

 name : name_string;
 root : access_root_template;
 parameters : access_parameter_list_element;
 access_procedures : access_access_procedure_list_element;
 IDAs : access_IDA_list_element;
END RECORD;

TYPE IDA(class : IDA_class_type) IS

RECORD

 name : name_string;
 file : file_name;
 template : access_IDA_template(class);
 data_types : access_data_type_list_element;
 CASE class IS
 WHEN pool => NULL;
 WHEN channel => parameters : access_parameter_list_element;
 END CASE;
END RECORD;

TYPE root_template IS

RECORD

 name : name_string;
 spec : file_name;
 bodie : file_name;
 parameters : access_parameter_list_element;
 access_procedures : access_access_procedure_list_element;
 data_types : access_data_type_list_element;
END RECORD;

TYPE IDA_template(class : IDA_class_type) IS

RECORD

 name : name_string;
 spec : file_name;
 bodie : file_name;
 access_procedures : access_access_procedure_list_element;
 CASE class IS
 WHEN pool => data_types : access_data_type_list_element;
 WHEN channel => parameters : access_parameter_list_element;
 END CASE;
END RECORD;

```

TYPE data_type IS
  RECORD
    name : name_string;
    spec : file_name;
    bodie : file_name;
  END RECORD;

TYPE node_class_type IS (act , channel , pool);
TYPE arc_class_type IS (from , to);

TYPE node(class : node_class_type);
TYPE access_node IS ACCESS node;
TYPE node_list_element;
TYPE access_node_list_element IS ACCESS node_list_element;

TYPE arc;
TYPE access_arc IS ACCESS arc;
TYPE arc_list_element;
TYPE access_arc_list_element IS ACCESS arc_list_element;

TYPE node IS
  RECORD
    name : name_string;
    arcs : access_arc_list_element;
    class : node_class_type;
  END RECORD;

TYPE node_list_element IS
  RECORD
    element : access_node;
    next : access_node_list_element := NULL;
  END RECORD;

TYPE arc IS
  RECORD
    name : name_string;
    direction : arc_class_type;
    source : access_node;
    sink : access_node;
  END RECORD;

TYPE arc_list_element IS
  RECORD
    element : access_arc;
    next : access_arc_list_element;
  END RECORD;

END construction_data_base_types;

```

```

WITH text_io, construction_data_base_types;
USE text_io, construction_data_base_types;

PROCEDURE construct_ACP_Diagram
    (head : OUT access_node_list_element)
IS
    tail          : access_node_list_element := NULL;
    work , work2 : access_node;
    last , last2 : natural;
    item , item2 : string(1 .. 80);
    class        : node_class_type;
    direction    : arc_class_type;
    accproc      : access_arc;

    PACKAGE node_type_io IS NEW enumeration_io(enum => node_class_type);
    PACKAGE arc_type_io  IS NEW enumeration_io(enum => arc_class_type);
    USE node_type_io , arc_type_io;

    PROCEDURE find_node(name : string ; act_node : OUT access_node);
    PROCEDURE append(act_node : IN OUT access_node;
                     proc      : IN OUT access_arc);

    PROCEDURE find_node(name : string ; act_node : OUT access_node)
    IS
        work : access_node_list_element;
    BEGIN
        work := head;
        LOOP
            EXIT WHEN work.element.name'last = name'last
                AND THEN
                work.element.name(1 .. work.element.name'last)
                    = name(1 .. name'last);
            work := work.next;
        END LOOP;
        act_node := work.element;
    END find_node;

    PROCEDURE append(act_node : IN OUT access_node;
                     proc      : IN OUT access_arc)
    IS
        work : access_arc_list_element;
    BEGIN
        work := act_node.arcs;
        IF work = NULL
        THEN
            act_node.arcs := NEW access_arc_list_element'(element => proc,
                                                            next    => NULL);
        ELSE
            WHILE work.next /= NULL
            LOOP
                work := work.next;
            END LOOP;
            work.next := NEW access_arc_list_element'(element => proc,
                                                        next    => NULL);
        END IF;
    END append;

BEGIN

```

```

head := NULL;

set_input(standard_input);
set_output(standard_output);

put_line("Start: Construction of ACP Diagram");

put_line("Start: Node Input");

LOOP
    put_line("Give Name of System Element");
    get_line(item , last);
    EXIT WHEN item(1) = '*';

    put_line("Give Class of System Element (act/pool/channel)");
    get(class); skip_line

    work := NEW access_node'(class => class,
                             name  => item(1 .. last),
                             arcs => NULL);

    IF head = NULL
    THEN
        head := NEW access_node_list_element'(element => work,
                                                next   => NULL);
        tail := head;
    ELSE
        tail.next := NEW access_node_list_element'(element => work,
                                                    next   => NULL);
        tail := tail.next;
    END IF;

END LOOP;

put_line("End: Node Input");

put_line("Start: Arc Input");

LOOP
    put_line("Give Name of Activity");
    get_line(item , last);
    EXIT WHEN item(1) = '*';

    find_node(item(1 .. last) , work);

    LOOP
        put_line("Give Name of Access Procedure");
        get_line(item , last);
        EXIT WHEN item(1) = '*';

        put_line("Give Direction (from/to)");
        get(direction); skip_line

        put_line("Give Name of IDA");
        get_line(item2 , last2);

        find_node(item2(1 .. last2) , work2);
    
```

```

    accproc := NEW access_arc'(name      => item(1 .. last),
                                direction => direction,
                                source    => work,
                                sink      => work2);

    append(work , accproc);
    append(work2 , accproc);

END LOOP;

END LOOP;

put_line("End: Arc Input");

put_line("End: Construction of ACP Diagram");

END construct_ACP_Diagram;

```



```

WITH construction_data_base_types;
USE construction_data_base_types;

PROCEDURE find_data_type
    (list : access_data_type_list_element;
     name : string;
     pointer : OUT access_data_type)
IS
    work : access_data_type_list_element;

BEGIN
    work := list;
    WHILE work /= NULL
    LOOP
        EXIT WHEN name'last = work.element.name'last
            AND THEN
                name(1..name'last) = work.element.name(1..name'last);
        work := work.next;
    END LOOP;

    IF work = NULL
    THEN
        pointer := NULL;
    ELSE
        pointer := work.element;
    END IF;

END find_data_type;

```

```

WITH construction_data_base_types;
USE construction_data_base_types;

PROCEDURE find_node
    (ACP_Diagram : access_node_list_element;
     class       : node_class_type;
     name        : string;
     pointer     : OUT access_node)
IS
    work : access_node_list_element;

BEGIN
    work := ACP_Diagram;
    WHILE work /= NULL
    LOOP
        EXIT WHEN class = work.element.class
            AND THEN
                (name'last = work.element.name'last
                 AND THEN
                     name(1..name'last) = work.element.name(1..name'last));
        work := work.next;
    END LOOP;

    IF work = NULL
    THEN
        pointer := NULL;
    ELSE
        pointer := work.element;
    END IF;
END find_node;

```

```

WITH construction_data_base_types;
USE  construction_data_base_types;

PROCEDURE find_root
    (list      : access_root_template_list_element;
     name      : string;
     pointer   : OUT access_root_template)
IS
    work : access_root_template_list_element;

BEGIN
    work := list;
    WHILE work /= NULL
    LOOP
        EXIT WHEN name'last = work.element.name'last
            AND THEN
                name(1..name'last) = work.element.name(1..name'last);
        work := work.next;
    END LOOP;

    IF work = NULL
    THEN
        pointer := NULL;
    ELSE
        pointer := work.element;
    END IF;

END find_root;

```

```

WITH construction_data_base_types;
USE construction_data_base_types;

PROCEDURE find_IDA_template
    (list      : access_IDA_template_list_element;
     name      : string;
     pointer   : OUT access_IDA_template)
IS
    work : access_IDA_template_list_element;

BEGIN
    work := list;
    WHILE work /= NULL
    LOOP
        EXIT WHEN name'last = work.element.name'last
            AND THEN
                name(1..name'last) = work.element.name(1..name'last);
        work := work.next;
    END LOOP;

    IF work = NULL
    THEN
        pointer := NULL;
    ELSE
        pointer := work.element;
    END IF;

END find_IDA_template;

```

```

WITH construction_data_base_types;
USE construction_data_base_types;

PROCEDURE match_parameters
    (list      : access_parameter_list_element;
     para_list : OUT access_parameter_list_element)
IS

    para_name : string(1 .. 80);
    para_last : natural;
    list_tail : access_parameter_list_element;
    tail      : access_parameter_list_element;

BEGIN

    list_tail := list;
    para_list := NULL;
    WHILE list_tail /= NULL
    LOOP
        put_line("Give Name of Actual Parameter for Formal Parameter '"
            & list_tail.element.formal(1..list_tail.element.formal'last)
            & "'");
        get_line(para_name , para_last);
        IF para_list = NULL
        THEN
            para_list :=
                NEW access_parameter_list_element'(
                    element => NEW access_parameter'(
                        formal => list_tail.element.formal
                            (1 ..
                                list_tail.element.formal'last),
                        actual => para_name(1 .. para_last)),
                    next    => NULL);
            tail := para_list;
        ELSE
            tail.next :=
                NEW access_parameter_list_element'(
                    element => NEW access_parameter'(
                        formal => list_tail.element.formal
                            (1 ..
                                list_tail.element.formal'last),
                        actual => para_name(1 .. para_last)),
                    next    => NULL);
            tail := tail.next;
        END IF;
        list_tail := list_tail.next;
    END LOOP;

END match_parameters;

```

```

WITH construction_data_base_types;
USE construction_data_base_types;

PROCEDURE match_procedures
    (list      : access_access_procedure_list_element;
     arc_list  : access_arc_list_element;
     proc_list : OUT access_access_procedure_list_element)
IS
    proc_name : string(1 .. 80);
    proc_last : natural;
    tail      : access_access_procedure_list_element;
    list_tail : access_access_procedure_list_element;
    arc_tail  : access_arc_list_element;

BEGIN
    list_tail := list;
    proc_list := NULL;

    WHILE list_tail /= NULL
    LOOP
        put_line("Give Name of Actual Access Procedure for Formal Procedure '"
            & list_tail.element.formal(1..list_tail.element.formal'last,
            & "'");
        get_line(proc_name , proc_last);
        arc_tail := arc_list;

        WHILE arc_tail /= NULL
        LOOP
            EXIT WHEN proc_last = arc_tail.element.name'last
                AND THEN
                proc_name(1..proc_last) = arc_tail.element.name
                    (1..proc_last);

            arc_tail := arc_tail.next;
        END LOOP;

        IF arc_tail = NULL
        THEN
            put_line("Actual Access Procedure with Name '"
                & proc_name(1 .. proc_last) & "' does not exist");
        ELSE
            IF proc_list = NULL
            THEN
                proc_list :=
                    NEW access_access_procedure_list_element'(
                        element => NEW access_access_procedure'(
                            formal => list_tail.element.formal
                                (1 ..
                                    list_tail.element.formal'last),
                            actual => proc_name(1..proc_last)),
                        next    => NULL);
                tail := proc_list;
            ELSE
                tail.next :=
                    NEW access_access_procedure_list_element'(
                        element => NEW access_access_procedure'(
                            formal => list_tail.element.formal

```

```

                                (1 ..
                                list_tail.element.formal'last),
                                actual => proc_name(1..proc_name)),
                                next    => NULL);
                                tail := tail.next;
                                END IF;

                                list_tail := list_tail.next;
                                END IF;
                                END LOOP;

                                END match_procedures;

```

```

WITH text_io, construction_data_base_types;
USE text_io, construction_data_base_types;

PROCEDURE form_data_types
    (data_types_list : OUT access_data_type_list_element)
IS
    name          : string(1 .. 80);
    last          : natural;
    answer        : character;
    tail         : access_data_type_list_element;
    list_element  : access_data_type;
BEGIN
    set_input(standard_input);
    set_output(standard_output);

    put_line("Start: Input Data Types");
    data_types_list := NULL;

    LOOP
        put_line("Give Name of Data Types");
        get_line(name , last);
        EXIT WHEN name(1) = '*';

        put_line("Is Data Type an Abstract Type? (Y/N)");
        get(answer); skip_line;
        CASE answer IS
            WHEN 'Y' | 'y' =>
                list_element :=
                    NEW access_data_type'(
                        name => name(1 .. last),
                        spec => "file." & name(1 .. last) & ".spec",
                        bodie => "file." & name(1 .. last) & ".body");
            WHEN OTHERS =>
                list_element :=
                    NEW access_data_type'(
                        name => name(1 .. last),
                        spec => "file." & name(1 .. last) & ".spec",
                        bodie => "");
        END CASE;

        IF data_types_list = NULL
        THEN
            data_types_list := NEW access_data_type_list_element'(
                element => list_element,
                next    => NULL);
            tail := data_types_list;
        ELSE
            tail.next := NEW access_data_type_list_element'(
                element => list_element,
                next    => NULL);
            tail := tail.next;
        END IF;
    END LOOP;

    put_line("End: Input Data Types");
END form_data_types;

```



```

WITH text_io, construction_data_base_types;
WITH find_data_type;
USE text_io, construction_data_base_types;

PROCEDURE form_IDA_templates
    (data_types_list : access_data_type_list_element;
     IDA_template_list : OUT access_IDA_template_list_element)
IS
    IDA_name      : string(1 .. 80);
    IDA_last      : natural;
    IDA_class     : IDA_class_type;
    proc_name     : string(1 .. 80);
    proc_last     : natural;
    data_name     : string(1 .. 80);
    data_last     : natural;
    access_data   : access_data_type;
    answer        : character;
    proc_list_head ,
    proc_list_tail      : access_access_procedure_list_element;
    channel_parameters_head ,
    channel_parameters_tail : access_parameter_list_element;
    tail                : access_IDA_template_list_element;
    IDA_element         : access_IDA_template;

    PACKAGE IDA_class_type_io IS NEW enumeration_io(IDA_class_type);
    USE IDA_class_type_io;

BEGIN
    set_input(standard_input);
    set_output(standard_output);

    put_line("Start: Input IDA Templates");
    IDA_template_list := NULL;

    LOOP
        put_line("Give Name of IDA Template");
        get_line(IDA_name , IDA_last);
        EXIT WHEN IDA_name(1) = '#';

        put_line("Give Class of IDA Template (pool/channel)");
        get(IDA_class); skip_line;

        put_line("Start: Input Access Procedure Names");
        proc_list_head := NULL;

        LOOP
            put_line("Give Name of Access Procedure");
            get_line(proc_name , proc_last);
            EXIT WHEN proc_name(1) = '#';

            IF proc_list_head = NULL
            THEN
                proc_list_head :=
                    NEW access_access_procedure_list_element'(
                        element => NEW access_access_procedure'(
                            formal => proc_name(1..proc_last),
                            actual => ""),
                        next      => NULL);
            END IF;
        END LOOP;
    END LOOP;

```

```

    proc_list_tail := proc_list_head;
ELSE
    proc_list_tail.next :=
        NEW access_access_procedure_list_element'(
            element => NEW access_access_procedure'(
                formal => proc_name(1..proc_last),
                actual => ""),
            next => NULL);
    proc_list_tail := proc_list_tail.next;
END IF;
END LOOP;

put_line("End: Input Access Procedure Names");

CASE IDA_class IS
    WHEN pool =>
        LOOP
            put_line("Give Name of Pool Data Type");
            get_line(data_name , data_last);
            find_data_type(data_types_list ,
                data_name(1 .. data_last) ,
                access_data);
            EXIT WHEN access_data /= NULL;
            put_line("There is no Data Type with Name '"
                & data_name(1 .. data_last) & "'");
        END LOOP;

    WHEN channel =>
        put_line("Start: Input Formal Channel Data Types");
        put_line("Is the Channel supposed to Work"
            & " on an Abstract Data Type? (Y/N)");
        get(answer); skip_line;

        CASE answer IS
            WHEN 'N' | 'n' =>
                put_line("The Formal Channel Data Type"
                    & "is 'channel_data'");
                channel_parameters_head :=
                    NEW access_parameter_list_element'(
                        element => NEW access_parameter'(
                            formal => "channel_data",
                            actual => ""),
                        next => NULL);

            WHEN OTHERS =>
                channel_parameters_head := NULL;
                LOOP
                    put_line("Give Name of Parameter");
                    get_line(data_name , data_last);
                    EXIT WHEN data_name(1) = ' ';

                    IF channel_parameters_head = NULL
                    THEN
                        channel_parameters_head :=
                            NEW access_parameter_list_element'(
                                element => NEW access_parameter'(
                                    formal => data_name
                                        (1..data_last),
                                    actual => ""),
                                next => NULL);

```

```

        channel_parameters_tail := channel_parameters_head;
    ELSE
        channel_parameters_tail.next :=
            NEW access_parameter_list_element'(
                element => NEW access_parameter'(
                    formal => data_name
                        (1..data_last),
                    actual => ""),
                next      => NULL);
        channel_parameters_tail := channel_parameters_tail.next;
    END IF;
END LOOP;
END CASE;

put_line("End: Input Formal Channel Data Types");

END CASE;

CASE IDA_class IS
    WHEN pool =>
        IDA_element :=
            NEW access_IDA_template'(
                class => pool,
                name  => IDA_name(1 .. IDA_last),
                spec  => "file." & IDA_name(1..IDA_last) & ".spec",
                bodie => "file." & IDA_name(1..IDA_last) & ".body",
                access_procedures => proc_list_head,
                data_types        => NEW access_data_type_list_element'(
                    element => access_data,
                    next     => NULL));

    WHEN channel =>
        IDA_element :=
            NEW access_IDA_template'(
                class => channel,
                name  => IDA_name(1 .. IDA_last),
                spec  => "file." & IDA_name(1..IDA_last) & ".spec",
                bodie => "file." & IDA_name(1..IDA_last) & ".body",
                access_procedures => proc_list_head,
                parameters        => channel_parameters_head);
END CASE;

IF IDA_template_list = NULL
THEN
    IDA_template_list := NEW access_IDA_template_list_element'(
        element => IDA_element,
        next    => NULL);

    tail := IDA_template_list;
ELSE
    tail.next := NEW access_IDA_template_list_element'(
        element => IDA_element,
        next    => NULL);

    tail := tail.next;
END IF;
END LOOP;

put_line("End: Input IDA Templates");

END form_IDA_templates;

```

```

WITH text_io, construction_data_base_types;
WITH find_data_type;
USE text_io, construction_data_base_types;

PROCEDURE form_root_templates
    (data_types_list : access_data_type_list_element;
     root_template_list : OUT access_root_template_list_element)
IS
    root_name      : string(1 .. 80);
    root_last      : natural;
    tail           : access_root_template_list_element;
    para_name      : string(1 .. 80);
    para_last      : natural;
    para_list_head ,
    para_list_tail : access_parameter_list_element;
    proc_name      : string(1 .. 80);
    proc_last      : natural;
    proc_list_head ,
    proc_list_tail : access_access_procedure_list_element;
    data_name      : string(1 .. 80);
    data_last      : natural;
    data_list_head ,
    data_list_tail : access_data_type_list_element;
    access_data     : access_data_type;
BEGIN
    set_input(standard_input);
    set_output(standard_output);

    put_line("Start: Input Root Templates");
    root_template_list := NULL;
    LOOP
        put_line("Give Name of Root Template");
        get_line(root_name , root_last);
        EXIT WHEN root_name(1) = '#';

        put_line("Start: Input Names of Formal Value Parameters");
        para_list_head := NULL;
        LOOP
            put_line("Give Name of Formal Value Parameter");
            get_line(para_name , para_last);
            EXIT WHEN para_name(1) = '#';

            IF para_list_head = NULL
            THEN
                para_list_head :=
                    NEW access_parameter_list_element'(
                        element => NEW access_parameter'(
                            formal => para_name(1..para_last),
                            actual => ""),
                        next      => NULL);
                para_list_tail := para_list_head;
            ELSE
                para_list_tail.next :=
                    NEW access_parameter_list_element'(
                        element => NEW access_parameter'(
                            formal => para_name(1..para_last),
                            actual => ""),
                        next      => NULL);
            END IF;
        END LOOP;
    END LOOP;

```

```

        para_list_tail := para_list_tail.next;
    END IF;
END LOOP;
put_line("End: Input Names of Formal Value Parameters");

put_line("Start: Input Names of Formal Access Procedures");
proc_list_head := NULL;
LOOP
    put_line("Give Name of Formal Access Procedure");
    get_line(proc_name , proc_last);
    EXIT WHEN proc_name(1) = '*';

    IF proc_list_head = NULL
    THEN
        proc_list_head :=
            NEW access_access_procedure_list_element'(
                element => NEW access_access_procedure'(
                    formal => proc_name(1..proc_last),
                    actual => ""),
                next    => NULL);
        proc_list_tail := proc_list_head;
    ELSE
        proc_list_tail.next :=
            NEW access_access_procedure_list_element'(
                element => NEW access_access_procedure'(
                    formal => proc_name(1..proc_last),
                    actual => ""),
                next    => NULL);
        proc_list_tail := proc_list_tail.next;
    END IF;
END LOOP;
put_line("End: Input Names of Formal Access Procedures");

put_line("Start: Input Names of Data Types");
data_list_head := NULL;
LOOP
    put_line("Give Name of Data Type");
    get_line(data_name , data_last);
    EXIT WHEN data_name(1) = '*';

    find_data_type(data_types_list ,
                    data_name(1 .. data_last) ,
                    access_data);
    IF access_data = NULL
    THEN
        put_line("There is no Data Type of Name '"
                & data_name(1 .. data_last) & "'");
    ELSE
        IF data_list_head = NULL
        THEN
            data_list_head :=
                NEW access_data_type_list_element'(
                    element => access_data,
                    next    => NULL);
            data_list_tail := data_list_head;
        ELSE
            data_list_tail.next :=
                NEW access_data_type_list_element'(
                    element => access_data,

```

```

        next      => NULL);
    data_list_tail := data_list_tail.next;
END IF;
END LOOP;
put_line("End: Input Names of Data Types");

IF root_template_list = NULL
THEN
    root_template_list :=
        NEW access_root_template_list_element'(
            element => NEW access_root_template'(
                name      => root_name(1..root_last),
                spec      => "file." &
                    root_name(1..root_last) &
                    ".spec",
                bodie     => "file." &
                    root_name(1..root_last) &
                    ".body",
                parameters => para_list_head,
                access_procedures => proc_list_head,
                data_types  => data_list_head),
            next      => NULL);
    tail := root_template_list;
ELSE
    tail.next :=
        NEW access_root_template_list_element'(
            element => NEW access_root_template'(
                name      => root_name(1..root_last),
                spec      => "file." &
                    root_name(1..root_last) &
                    ".spec",
                bodie     => "file." &
                    root_name(1..root_last) &
                    ".body",
                parameters => para_list_head,
                access_procedures => proc_list_head,
                data_types  => data_list_head),
            next      => NULL);
    tail := tail.next;
END IF;
END LOOP;

put_line("End: Input Root Templates");

END form_root_templates;

```

```

WITH text_io, construction_data_base_types;
WITH find_data_type , find_node , find_root , find_IDA_template;
WITH match_parameters , match_procedures;
USE text_io, construction_data_base_types;

```

PROCEDURE form_Subsystems

```

    (data_types_list      : access_data_type_list_element;
     IDA_template_list    : access_IDA_template_list_element;
     root_template_list   : access_root_template_list_element;
     ACP_Diagram          : access_node_list_element;
     subsystem_list       : OUT access_subsystem_list_element)

```

IS

```

    answer                : character;
    subsystem_name        : string(1 .. 80);
    subsystem_last        : natural;
    tail                  : access_subsystem_list_element;
    para_list_head        : access_parameter_list_element;
    act_list_head         : access_activity_list_element;
    subsystem_IDA_list_head ,
    IDA_list_head         : access_IDA_list_element;
    subsystem_object      : access_subsystem;

```

```

    PROCEDURE get_parameters(head : OUT access_parameter_list_element);

```

PROCEDURE get_activities

```

    (ACP_Diagram          : access_node_list_element;
     subsystem_list       : access_subsystem_list_element;
     root_list            : access_root_template_list_element;
     head                 : OUT access_activity_list_element);

```

PROCEDURE get_IDAs

```

    (ACP_Diagram          : access_node_list_element;
     act_head             : access_activity_list_element;
     template_list        : access_IDA_template_list_element;
     IDA_head             : OUT access_IDA_list_element;
     sub_IDA_head         : OUT access_IDA_list_element;
     data_types_list      : access_data_type_list_element);

```

PROCEDURE add_IDAs_to_activity

```

    (ACP_Diagram          : access_node_list_element;
     act_head             : IN OUT access_activity_list_element;
     IDA_head             : access_IDA_list_element;
     sub_IDA_head         : access_IDA_list_element);

```

```

    PROCEDURE get_parameters(head : OUT access_parameter_list_element)

```

IS

```

    tail                  : access_parameter_list_element;
    para_name             : string(1 .. 80);
    para_last             : natural;

```

BEGIN

```

    put_line("Start: Input Name of Formal Value Parameters");
    head := NULL;

```

LOOP

```

    put_line("Give Name of Value Parameter");
    get_line(para_name , para_last);
    EXIT WHEN para_name(1) = '*';

```

```

IF head = NULL
THEN
    head := NEW access_parameter_list_element'(
        element => NEW access_parameter'(
            formal => para_name(1..para_last),
            actual => ""),
        next    => NULL);
    tail := head;
ELSE
    tail.next :=
        NEW access_parameter_list_element'(
            element => NEW access_parameter'(
                formal => para_name(1..para_last),
                actual => ""),
            next    => NULL);
    tail := tail.next;
END IF;
END LOOP;

put_line("End: Input Names of Formal Value Parameters");

END get_parameters;

PROCEDURE get_activities
    (ACP_Diagram      : access_node_list_element;
     subsystem_list   : access_subsystem_list_element;
     root_list        : access_root_template_list_element;
     head             : OUT access_activity_list_element)
IS
    tail           : access_activity_list_element;
    act_name       : string(1 .. 80);
    act_last       : natural;
    root_name      : string(1 .. 80);
    root_last      : natural;
    act_object     : access_activity;
    access_act     : access_node;
    access_root    : access_root_template;
    para_list      : access_parameter_list_element;
    proc_list      : access_access_procedure_list_element;

    FUNCTION test_used
        (name          : string;
         subsystem_list : access_subsystem_list_element)
    RETURN boolean
    IS
        subsystem_tail : access_subsystem_list_element;
        act_tail        : access_activity_list_element;
    BEGIN
        subsystem_tail := subsystem_list;
        WHILE subsystem_tail /= NULL
        LOOP
            act_tail := subsystem_tail.activities;
            WHILE act_tail /= NULL
            LOOP
                IF name'last = act_tail.element.name'last
                AND THEN
                    name(1..name'last) = act_tail.element.name(1..name'last)
                THEN
                    RETURN true;
                END IF;
            END LOOP;
            subsystem_tail := act_tail.subsystem_list;
        END LOOP;
    END FUNCTION;

```



```

        END IF;
        act_tail := act_tail.next;
    END LOOP;
    subsystem_tail := subsystem_tail.next;
END LOOP;
RETURN false;
END test_used;

```

BEGIN

```

put_line("Start: Form Activities");
head := NULL;

```

LOOP

```

    put_line("Give Name of Activity");
    get_line(act_name , act_last);
    EXIT WHEN act_name(1) = '#';

```

```

    IF test_used(act_name(1..act_last) , subsystem_list)
    THEN

```

```

        put_line("Activity '" & act_name(1..act_last)
                & "' is already used");

```

ELSE

```

        find_node(ACP_Diagram , act ,
                act_name(1..act_last) , access_act);

```

```

        IF access_act = NULL

```

THEN

```

            put_line("Activity '" & act_name(1..act_last)
                    & "' does not exist");

```

ELSE

LOOP

```

        put_line("Give Name of Root Template");
        get_line(root_name , root_last);
        find_root(root_list , root_name(1..root_last) ,
                access_root);
        EXIT WHEN access_root /= NULL;
        put_line("Root Template '" & root_name(1..root_last)
                & "' does not exist");

```

END LOOP;

act_object :=

```

    NEW access_activity'(
        name           => act_name(1 .. act_last);
        root           => access_root,
        parameters     => NULL,
        access_procedures => NULL,
        IDAs           => NULL);

```

```

match_parameters(access_root.parameters , para_list);
act_object.parameters := para_list;

```

```

match_procedures(access_root.access_procedures ,
                access_act.arcs , proc_list);
act_object.access_procedures := proc_list;

```

IF head = NULL

THEN

```

    head := NEW access_activity_list_element'(
        element => act_object,
        next    => NULL);

```

tail := head;

```

        ELSE
            tail.next := NEW access_activity_list_element'(
                element => act_object,
                next    => NULL);
            tail := tail.next;
        END IF;
    END IF;
END LOOP;

put_line("End: Form Activities");

END get_activities;

PROCEDURE get_IDAs
    (ACP_Diagram      : access_node_list_element;
     act_head         : access_activity_list_element;
     template_list    : access_IDA_template_list;
     IDA_head         : OUT access_IDA_list_element;
     sub_IDA_head     : OUT access_IDA_list_element;
     data_types_list  : access_data_type_list_element)
IS
    act_tail          : access_activity_list_element;
    access_act        : access_node;
    arc_head          : access_arc_list_element;
    access_IDA_node    : access_node;
    IDA_name          : string(1 .. 80);
    IDA_last          : natural;
    access_template    : access_IDA_template;
    IDA_object         : access_IDA;
    IDA_tail ,
    sub_IDA_tail       : access_IDA_list_element;
    para_list         : access_parameter_list_element;
    data_name          : string(1 .. 80);
    data_last         : natural;
    access_data        : access_data_type;

    FUNCTION test_used
        (name          : string;
         IDA_list       : access_IDA_list_element;
         sub_IDA_list   : access_IDA_list_element)
    RETURN boolean
    IS
        work : access_IDA_list_element;
    BEGIN
        work := IDA_list;
        WHILE work /= NULL
        LOOP
            IF name'last = work.element.name'last
            AND THEN
                name(1..name'last) = work.element.name(1..name'last)
            THEN
                RETURN true;
            END IF;
            work := work.next;
        END LOOP;
        work := sub_IDA_list;
        WHILE work /= NULL
        LOOP

```

```

        IF name'last = work.element.name'last
            AND THEN
                name(1..name'last) = work.element.name(1..name'last)
            THEN
                RETURN true;
            END IF;
            work := work.next;
        END LOOP;
        RETURN false;
    END test_used;

```

```

FUNCTION test_sub_IDA
    (act_list : access_activity_list_element;
     IDA_node : access_node)
RETURN boolean
IS
    act_tail : access_activity_list;
    arc_list : access_arc_list_element;
BEGIN
    arc_list := IDA_node.arcs;
    WHILE arc_list /= NULL
        LOOP
            act_tail := act_list;
            WHILE act_tail /= NULL
                LOOP
                    EXIT WHEN act_tail.element.name'last
                        = arc_list.element.source.name'last
                        AND THEN
                            act_tail.element.name(1..act_tail.element.name'last)
                                = arc_list.element.source.name
                                    (1..arc_list.element.source.name'last);
                    act_tail := act_tail.next;
                END LOOP;
            IF act_tail = NULL
                THEN
                    RETURN true;
                END IF;
            arc_list := arc_list.next;
        END LOOP;
        RETURN false;
    END test_sub_IDA;

```

```

BEGIN
    put_line("Start: Forming IDAs");
    act_tail := act_head;
    IDA_head := NULL;
    sub_IDA_head := NULL;
    WHILE act_tail /= NULL
        LOOP
            find_node(ACP_Diagram , act ,
                act_tail.element.name(1..act_tail.element.name'last) ,
                access_act);
            arc_head := access_act.arcs;
            WHILE arc_head /= NULL
                LOOP
                    access_IDA_node := arc_head.element.sink;
                    IF NOT test_used(access_IDA_node.name(1..access_IDA_node.name'last),
                        IDA_head , sub_IDA_head)
                        THEN

```

```

LOOP
  LOOP
    put_line("Give Name of IDA Template for IDA '"
      & access_IDA_node.name(1..access_IDA_node.name'last)
      & "'");
    get_line(IDA_name , IDA_last);
    find_IDA_template(template_list ,
      IDA_name(1 .. IDA_last) ,
      access_template);
    EXIT WHEN access_template /= NULL;
    put_line("IDA Template '" & IDA_name(1..IDA_last)
      & "' does not exist");
  END LOOP;
  EXIT WHEN ((access_IDA_node.class = pool
    AND access_template.class = pool)
    OR
    (access_IDA_node.class = channel
    AND IDA_template.class = channel));
  put_line("Class of IDA and IDA Template do not match");
  END LOOP;

  IF test_sub_IDA(act_head , access_IDA_node)
  THEN
    CASE access_IDA_node.class IS
      WHEN pool =>
        IDA_object :=
          NEW access_IDA'(
            class => pool,
            name  => access_IDA_node.name
              (1..access_IDA_node.name'last),
            file  => "file." &
              access_IDA_node.name
              (1..access_IDA_node.name'last),
            template  => access_template,
            data_types => access_template.data_types);
      WHEN channel =>
        match_parameters(access_template.parameters ,
          para_list);
        LOOP
          put_line("Give Name of Data Type for Channel '"
            & access_IDA_node.name
              (1..access_IDA_node.name'last)
            & "'");
          get_line(data_name , data_last);
          find_data_type(data_types_list ,
            data_name(1..data_last) ,
            access_data);
          EXIT WHEN access_data /= NULL;
          put_line("Data Type '" & data_name(1..data_last)
            & "' does not exist");
        END LOOP;
        IDA_object :=
          NEW access_IDA'(
            class => channel,
            name  => access_IDA_node.name
              (1..access_IDA_node.name'last),
            file  => "file." &
              access_IDA_node.name
              (1..access_IDA_node.name'last),

```

```

        template => access_template,
        data_types =>
            NEW access_data_type_list_element'(
                element => access_data,
                next => NULL),
        parameters => para_list);
END CASE;
IF sub_IDA_head = NULL
THEN
    sub_IDA_head :=
        NEW access_IDA_list_element'(
            element => IDA_object,
            next => NULL);
    sub_IDA_tail := sub_IDA_head;
ELSE
    sub_IDA_tail.next :=
        NEW access_IDA_list_element'(
            element => IDA_object,
            next => NULL);
    sub_IDA_tail := sub_IDA_tail.next;
END IF;
ELSE
CASE access_IDA_node.class IS
    WHEN pool =>
        IDA_object :=
            NEW access_IDA'(
                class => pool,
                name => access_IDA_node.name
                    (1..access_IDA_node.name'last),
                file => "",
                template => access_template,
                data_types => access_template.data_types);
    WHEN channel =>
        match_parameters(access_template.parameters ,
            para_list);
    LOOP
        put_line("Give Name of Data Type for Channel '"
            & access_IDA_node.name
                (1..access_IDA_node.name'last)
            & "'");
        get_line(data_name , data_last);
        find_data_type(data_types_list ,
            data_name(1..data_last) ,
            access_data);
        EXIT WHEN access_data /= NULL;
        put_line("Data Type '" & data_name(1..data_last)
            & "' does not exist");
    END LOOP;
    IDA_object :=
        NEW access_IDA'(
            class => channel,
            name => access_IDA_node.name
                (1..access_IDA_node.name'last),
            file => "",
            template => access_template,
            data_types =>
                NEW access_data_type_list_element'(
                    element => access_data,
                    next => NULL),

```

```

                                parameters => para_list));
END CASE;
IF IDA_head = NULL
THEN
    IDA_head := NEW access_IDA_list_element'(
                                element => IDA_object,
                                next    => NULL);
    IDA_tail := IDA_head;
ELSE
    IDA_tail.next := NEW access_IDA_list_element'(
                                element => IDA_object,
                                next    => NULL);
    IDA_tail := IDA_tail.next;
END IF;
END IF;
arc_head := arc_head.next;
END LOOP;
act_tail := act_tail.next;
END LOOP;

put_line("End: Forming IDAs");

END get_IDAs;

PROCEDURE add_IDAs_to_activity
(ACP_diagram : access_node_list_element;
 act_head    : IN OUT access_activity_list_element;
 IDA_head    : access_IDA_list_element;
 sub_IDA_head : access_IDA_list_element)
IS
    act_tail      : access_activity_list_element;
    access_act     : access_node;
    arc_head      : access_arc_list_element;
    act_IDA_head ,
    act_IDA_tail  : access_IDA_list_element;
    access_act_IDA : access_IDA;

    PROCEDURE find_IDA
    (IDA_head      : access_IDA_list_element;
     sub_IDA_head : access_IDA_list_element;
     name         : string;
     pointer      : access_IDA)
    IS
        work : access_IDA_list_element;
    BEGIN
        work := IDA_head;
        WHILE work /= NULL
        LOOP
            EXIT WHEN name'last = work.element.name'last
            AND THEN
                name(1..name'last) = work.element.name(1..name'last);
            work := work.next;
        END LOOP;
        IF work /= NULL
        THEN
            pointer := work.element;
        ELSE
            work := sub_IDA_head;
        END IF;
    END find_IDA;

```

```

        WHILE work /= NULL
        LOOP
            EXIT WHEN name'last = work.element.name'last
            AND THEN
                name(1..name'last) = work.element.name(1..name'last);
            work := work.next;
        END LOOP;
        IF work /= NULL
        THEN
            pointer := work.element;
        ELSE
            pointer := NULL;
        END IF;
    END IF;
END find_IDA;

```

```

BEGIN
    act_tail := act_head;
    WHILE act_tail /= NULL
    LOOP
        find_node(ACP_Diagram , act ,
            act_tail.element.name(1..act_tail.element.name'last) ,
            access_act);
        arc_head := access_act.arcs;
        act_IDA_head := NULL;
        WHILE arc_head /= NULL
        LOOP
            find_IDA(IDA_head , sub_IDA_head ,
                arc_head.element.sink.name
                    (1..arc_head.element.sink.name'last) ,
                access_act_IDA);
            IF act_IDA_head = NULL
            THEN
                act_IDA_head := NEW access_IDA_list_element'(
                    element => access_act_IDA,
                    next    => NULL);
                act_IDA_tail := act_IDA_head;
            ELSE
                act_IDA_tail.next := NEW access_IDA_element_list'(
                    element => access_act_IDA,
                    next    => NULL);
                act_IDA_tail := act_IDA_tail.next;
            END IF;
        END LOOP;
        act_tail.element.IDAs := act_IDA_head;
        act_tail := act_tail.next;
    END LOOP;
END add_IDAs_to_activity;

```

```

BEGIN
    set_input(standard_input);
    set_output(standard_output);

    put_line("Start: Forming of Subsystems");
    subsystem_list := NULL;
    put_line("Are Subsystems Required? (Y/N)");
    get(answer); skip_line;

    CASE answer IS

```

```

WHEN 'N' | 'n' =>
    NULL;
WHEN OTHERS =>
    LOOP
        put_line("Give Name of Subsystem");
        get_line(subsystem_name , subsystem_last);
        EXIT WHEN subsystem_name(1) = '#';

        subsystem_object :=
            NEW access_subsystem'(
                name => subsystem_name(1..subsystem_last),
                file => "file." & subsystem_name(1..subsystem_last),
                parameters => NULL,
                activities => NULL,
                IDAs => NULL,
                subsystem_IDAs => NULL);

        get_parameters(para_list_head);
        subsystem_object.parameters := para_list_head;

        get_activities(ACP_Diagram ,
            subsystem_list ,
            root_template_list ,
            act_list_head);

        get_IDAs(ACP_Diagram ,
            act_list_head ,
            IDA_template_list ,
            IDA_list_head ,
            subsystem_IDA_list_head ,
            data_types_list);
        subsystem_object.IDAs := IDA_list_head;
        subsystem_object.subsystem_IDAs := subsystem_IDA_list_head;

        add_IDAs_to_activity(ACP_Diagram ,
            act_list_head ,
            IDA_list_head ,
            subsystem_IDA_list_head);
        subsystem_object.activities := act_list_head;

        IF subsystem_list = NULL
        THEN
            subsystem_list := NEW access_subsystem_list_element'(
                element => subsystem_object,
                next => NULL);

            tail := subsystem_list;
        ELSE
            tail.next := NEW access_subsystem_list_element'(
                element => subsystem_object,
                next => NULL);

            tail := tail.next;
        END IF;
    END LOOP;
END CASE;

put_line("End: Forming of Subsystems");

END form_Subsystems;

```



```

WITH text_io, construction_data_base_types;
WITH match_parameters , match_procedures;
WITH find_IDA_template , find_data_type , find_root;
USE text_io, construction_data_base_types;

PROCEDURE form_MASCOT_System
    (data_types_list      : access_data_type_list_element;
     IDA_template_list    : access_IDA_template_list_element;
     root_template_list   : access_root_template_list_element;
     subsystem_list       : IN OUT access_subsystem_list_element;
     ACP_Diagram          : access_node_list_element;
     MASCOT_System_object : OUT access_MASCOT_System)

IS
    sys_name : string(1 .. 80);
    sys_last : natural;
BEGIN
    set_input(standard_input);
    set_output(standard_output);
    put_line("Give Name of Main Program");
    get_line(sys_name , sys_last);

    IF subsystem_list /= NULL
    THEN
        DECLARE
            subsystem_tail : access_subsystem_list_element;
            para_list      : access_parameter_list_element;
            sub_IDA_list    : access_IDA_list_element;

            PROCEDURE append_sub_IDAs
                (IDA_list      : access_IDA_list_element;
                 sub_IDA_list : IN OUT access_IDA_list_element)
            IS
                tail , work , work_tail : access_IDA_list_element;
            BEGIN
                IF sub_IDA_list = NULL
                THEN
                    sub_IDA_list := IDA_list;
                ELSE
                    tail := sub_IDA_list;
                    WHILE tail.next /= NULL
                    LOOP
                        tail := tail.next;
                    END LOOP;
                    work := IDA_list;
                    WHILE work /= NULL
                    LOOP
                        work_tail := sub_IDA_list;
                        WHILE work_tail /= NULL
                        LOOP
                            EXIT WHEN work.element.name'last
                                = work_tail.element.name'last
                                AND THEN
                                    work.element.name(1..work.element.name'last)
                                        = work_tail.element.name
                                            (1..work_tail.element.name'last);
                            work_tail := work_tail.next;
                        END LOOP;
                        IF work_tail = NULL

```

```

        THEN
            tail.next := NEW access_IDA_list_element'(
                element => work.element,
                next    => NULL);
            tail := tail.next;
        END IF;
        work := work.next;
    END LOOP;
END IF;
END append_sub_IDAs;
BEGIN
    put_line("Start: Forming of Main Program with Subsystems");

    sub_IDA_list := NULL;
    subsystem_tail := subsystem_list;
    WHILE subsystem_tail /= NULL
    LOOP
        put_line("Subsystem '"
            & subsystem_tail.element.name
            (1..subsystem_tail.element.name'last)
            & "': Input Actual Value Parameters");
        match_parameters(subsystem_tail.element.parameters ,
            para_list);
        subsystem_tail.element.parameters := para_list;
        append_sub_IDAs(subsystem_tail.element.subsystem_IDAs ,
            sub_IDA_list);
        subsystem_tail := subsystem_tail.next;
    END LOOP;

    MASCOT_System_object :=
        NEW access_MASCOT_System'(
            class      => with_subsystems,
            name       => sys_name(1 .. sys_last),
            file       => "file." & sys_name(1 .. sys_last),
            subsystems => subsystem_list,
            subsystem_IDAs => sub_IDA_list);

    put_line("End: Forming of Main Program with Subsystems");
END;
ELSE
    DECLARE
        act_head , act_tail : access_activity_list_element;
        IDA_head , IDA_tail : access_IDA_list_element;
        node_tail          : access_node_list_element;
        template_name      : string(1 .. 80);
        template_last      : natural;
        access_template     : access_IDA_template;
        access_root         : access_root_template;
        para_list           : access_parameter_list_element;
        data_name           : string(1 .. 80);
        data_last          : natural;
        access_data         : access_data_type;
        proc_list           : access_access_procedure_list_element;
        arc_tail            : access_arc_list_element;
        act_IDA_head ,
        act_IDA_tail       : access_IDA_list_element;
    BEGIN
        put_line("Start: Forming of Main Program without Subsystems");

```

```

put_line("Start: Forming IDAs");
node_tail := ACP_Diagram;
IDA_head := NULL;
WHILE node_tail /= NULL
LOOP
  IF node_tail.element.class /= act
  THEN
    CASE node_tail.element.class IS
      WHEN pool =>
        LOOP
          LOOP
            put_line("Give Name of Template for Pool '"
                      & node_tail.element.name
                      (1 .. node_tail.element.name'last)
                      & "'");
            get_line(template_name , template_last);
            find_IDA_template(IDA_template_list ,
                              template_name(1..template_last) ,
                              access_template);
            EXIT WHEN access_template /= NULL;
            put_line("IDA Template '"
                      & template_name(1 .. template_last)
                      & "' does not exist");
          END LOOP;
          EXIT WHEN access_template.class = pool;
          put_line("IDA Template '"
                    & template_name(1 .. template_last)
                    & "' is not a Pool Template");
        END LOOP;
      IF IDA_head = NULL
      THEN
        IDA_head :=
          NEW access_IDA_list_element'(
            element =>
              NEW access_IDA'(
                class      => pool,
                name       =>
                  node_tail.element.name(1..node_tail.element.name'last),
                file       => "",
                template   => access_template,
                data_types =>
                  access_template.data_types),
            next          => NULL);
        IDA_tail := IDA_head;
      ELSE
        IDA_tail.next :=
          NEW access_IDA_list_element'(
            element =>
              NEW access_IDA'(
                class      => pool,
                name       =>
                  node_tail.element.name(1..node_tail.element.name'last),
                file       => "",
                template   => access_template,
                data_types =>
                  access_template.data_types),
            next          => NULL);
        IDA_tail := IDA_tail.next;
      END IF;
    END LOOP;
  END IF;

```

```

WHEN channel =>
  LOOP
    LOOP
      put_line("Give Name of Template for Channel '"
        & node_tail.element.name
        (1 .. node_tail.element.name'last)
        & "'");
      get_line(template_name , template_last);
      find_IDA_template(IDA_template_list ,
        template_name(1..template_last) ,
        access_template);
      EXIT WHEN access_template /= NULL;
      put_line("IDA Template '"
        & template_name(1 .. template_last)
        & "' does not exist");
    END LOOP;
    EXIT WHEN access_template.class = channel;
    put_line("IDA Template '"
      & template_name(1 .. template_last)
      & "' is not a Channel Template");
  END LOOP;
  put_line("Give Actual Types for Formal Data Types");
  match_parameters(access_template.parameters ,
    para_list);
  LOOP
    put_line("Give Name of Data Type");
    get_line(data_name , data_last);
    find_data_type(data_types_list ,
      data_name(1 .. data_last) ,
      access_data);
    EXIT WHEN access_data /= NULL;
    put_line("Data Type '" & data_name(1..data_last)
      & "' does not exist");
  END LOOP;
  IF IDA_head = NULL
  THEN
    IDA_head :=
      NEW access_IDA_list_element'(
        element =>
          NEW access_IDA'(
            class      => channel,
            name       =>
              node_tail.element.name(1..node_tail.element.name'last),
            file       => "",
            template   => access_template,
            data_types =>
              NEW access_data_type_list_element'(
                element => access_data,
                next    => NULL),
            parameters => para_list),
        next    => NULL);
    IDA_tail := IDA_head;
  ELSE
    IDA_tail.next :=
      NEW access_IDA_list_element'(
        element =>
          NEW access_IDA'(
            class      => channel,
            name       =>

```

```

        node_tail.element.name(1..node_tail.element.name'last),
                                file      => "",
                                data_types =>
                                    NEW access_data_type_list_element'(
                                        element => access_data,
                                        next    => NULL);
                                parameters => para_list),
                                next      => NULL);
        IDA_tail := IDA_tail.next;
    END IF;
END CASE;
END IF;
node_tail := node_tail.next;
END LOOP;
put_line("End: Forming IDAs");

put_line("Start: Forming Activities")
node_tail := ACP_Diagram;
act_head := NULL;
WHILE node_tail /= NULL
LOOP
    IF node_tail.element.class = act
    THEN
        LOOP
            put_line("Give Name of Template for Activity '"
                    & node_tail.element.name
                    (1..node_tail.element.name'last)
                    & "'");
            get_line(template_name , template_last);
            find_root(root_template_list ,
                    template_name(1 .. template_last) ,
                    access_root);
            EXIT WHEN access_root /= NULL;
            put_line("Root Template '" & template_name(1..template_last)
                    & "' does not exist");
        END LOOP;
        put_line("Give Actual Value Parameters")
        match_parameters(access_root.parameters , para_list);
        put_line("Give Names of Actual Access Procedures");
        match_procedures(access_root.access_procedures ,
                node_tail.element.arcs ,
                proc_list);
        act_IDA_head := NULL;
        arc_tail := node_tail.element.arcs;
        WHILE arc_tail /= NULL
        LOOP
            IDA_tail := IDA_head;
            WHILE IDA_tail /= NULL
            LOOP
                EXIT WHEN arc_tail.element.sink.name'last
                        = IDA_tail.element.name'last
                AND THEN
                    arc_tail.element.sink.name
                    (1 .. arc_tail.element.sink.name'last)
                    = IDA_tail.element.name
                    (1 .. IDA_tail.element.name'last);
                IDA_tail := IDA_tail.next;
            END LOOP;
            IF act_IDA_head = NULL

```

```

THEN
    act_IDA_head := NEW access_IDA_list_element'(
        element => IDA_tail.element,
        next    => NULL);
    act_IDA_tail := act_IDA_head;
ELSE
    act_IDA_tail.next := NEW access_IDA_list_element'(
        element => IDA_tail.element,
        next    => NULL);
    act_IDA_tail := act_IDA_tail.next;
END IF;
arc_tail := arc_tail.next;
END LOOP;
IF act_head = NULL
THEN
    act_head :=
        NEW access_activity_list_element'(
            element =>
                NEW access_activity'(
                    name          =>
                        node_tail.element.name(1..node_tail.element.name'last),
                    root          => access_root,
                    parameters    => para_list,
                    access_procedures => proc_list,
                    IDAs          => act_IDA_head),
            next    => NULL);
    act_tail := act_head;
ELSE
    act_tail.next :=
        NEW access_activity_list_element'(
            element =>
                NEW access_activity'(
                    name          =>
                        node_tail.element.name(1..node_tail.element.name'last),
                    root          => access_root,
                    parameters    => para_list,
                    access_procedures => proc_list,
                    IDAs          => act_IDA_head),
            next    => NULL);
    act_tail := act_tail.next;
END IF;
END IF;
node_tail := node_tail.next;
END LOOP;
put_line("End: Forming Activities");

MASCOT_System_object :=
    NEW access_MASCOT_System_object'(
        class    => without_subsystems,
        name     => sys_name(1 .. sys_last),
        file     => "file." & sys_name(1 .. sys_last),
        activities => act_head,
        IDAs     => IDA_head);

put_line("End: Forming of Main Program without Subsystems");
END;
END IF;
END form_MASCOT_System;

```

```

WITH construction_data_base_types;
WITH construct_ACP_Diagram;
WITH form_MASCOT_System;
WITH form_Subsystems;
WITH form_data_types;
WITH form_IDA_templates;
WITH form_root_templates;
WITH generate_data_types_frame, generate_root_templates_frame,
    generate_IDA_templates_frame, generate_subsystem_frame,
    generate_main_program_frame;
WITH save_MASCOT_System;
WITH save_ACP_Diagram;
USE construction_data_base_types;

PROCEDURE construct_data_base
IS
    ACP_Diagram          : access_node_list_element;
    MASCOT_System_object : access_MASCOT_System;
    data_types_list      : access_data_type_list_element;
    IDA_template_list    : access_IDA_template_list_element;
    root_template_list   : access_root_template_list_element;
    subsystem_list       : access_subsystem_list_element;
BEGIN
    construct_ACP_Diagram(ACP_Diagram);

    form_data_types(data_types_list);
    generate_data_types_frame(data_types_list);

    form_IDA_templates(data_types_list , IDA_template_list);
    generate_IDA_templates_frame(IDA_template_list);

    form_root_templates(data_types_list , root_template_list);
    generate_root_templates_frame(root_template_list);

    form_Subsystems(data_types_list ,
                    IDA_template_list ,
                    root_template_list ,
                    ACP_Diagram ,
                    subsystem_list);
    generate_subsystem_frame(subsystem_list);

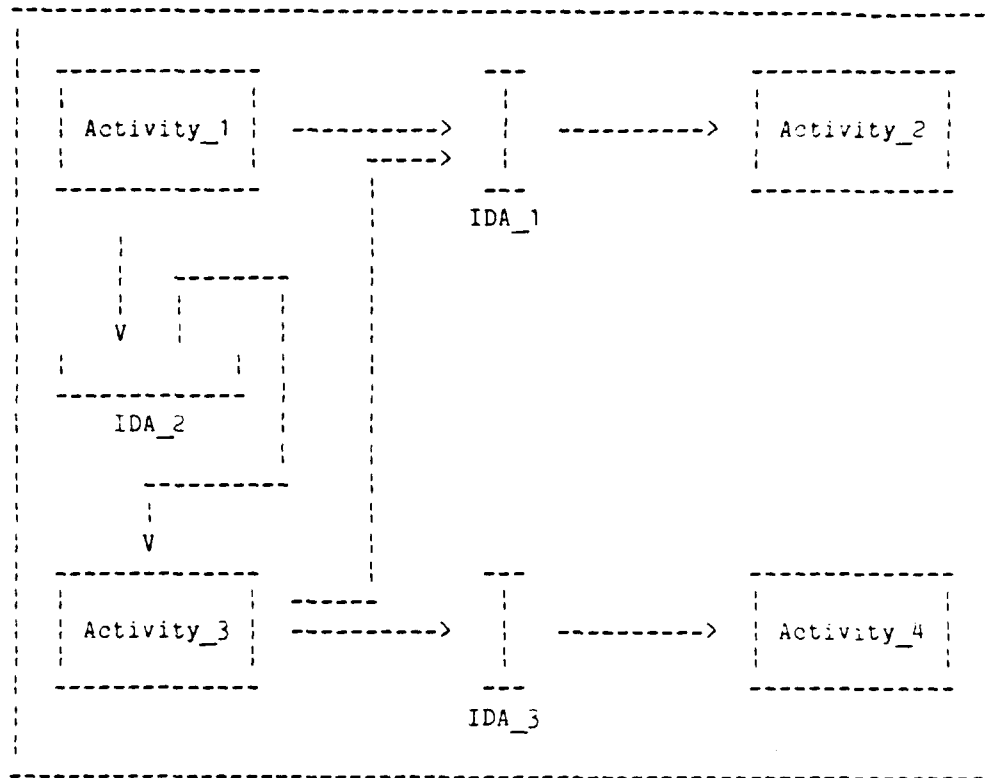
    form_MASCOT_System(data_types_list ,
                    IDA_template_list ,
                    root_template_list ,
                    subsystem_list ,
                    ACP_Diagram ,
                    MASCOT_System_object);
    generate_main_program_frame(MASCOT_System_object);

    save_MASCOT_System(MASCOT_System_object , data_types_list ,
                    IDA_template_list , root_template_list);
    save_ACP_Diagram(ACP_Diagram);
END construct_data_base;

```

1. The ACP Diagram

The use of the MASCOT philosophy in an Ada environment is demonstrated by a simple example. Its ACP Diagram is shown below.



Activity 1 generates a string and either puts this string into Pool IDA_2 or sends a number to Activity 2. Activity 3 reads from Pool IDA_2 and sends a number either to Activity 2 or 4 depending on the value of a constant parameter. Activities 2 and 4 only consume the messages. The codes are equivalent.

IDA_2 stores the strings generating a key with which the strings can be retrieved. Readers of IDA_2 are given precedence over writers. Activity 3 generates a key randomly to retrieve the strings stored in IDA_2. IDA_1 is a fast channel storing several data objects temporarily. IDA_3 is a very slow channel. It can only hold one data object at a time. The data objects passed through IDA 1 and 3 are equivalent.

The ACP Diagram is transferred by the procedure "construct_ACP_Diagram" into a chained list shown below. For easier reading the form of the list and of its elements is not totally equivalent to the Ada implementation. Some necessary references of the Ada data structure are dropped for this reason. A reference is indicated by the sign "->". Strings are enclosed in quotes.


```

ACP_Diagram: element: -> node_1
              next:    --
              |
              -----
              |
-->: element: -> node_2
              next:    --
              |
              -----
              |
-->: element: -> node_3
              next:    --
              |
              -----
              |
-->: element: -> node_4
              next:    --
              |
              -----
              |
-->: element: -> node_5
              next:    --
              |
              -----
              |
-->: element: -> node_6
              next:    --
              |
              -----
              |
-->: element: -> node_7
              next:    NULL

```

```

arc_list_1: element: -> arc_1
            next:    --
            |
            -----
            |
-->: element: -> arc_2
            next:    NULL

arc_list_3: element: -> arc_3
            next:    NULL

```

```

node_1: name: "Activity_1"
      arcs: -> arc_list_1
      class: act

node_2: name: "IDA_1"
      arcs: -> arc_list_2
      class: channel

node_3: name: "Activity_2"
      arcs: -> arc_list_3
      class: act

node_4: name: "IDA_2"
      arcs: -> arc_list_4
      class: pool

node_5: name: "Activity_3"
      arcs: -> arc_list_5
      class: act

node_6: name: "IDA_3"
      arcs: -> arc_list_6
      class: channel

node_7: name: "Activity_4"
      arcs: -> arc_list_7
      class: act

```

```

arc_list_2: element: -> arc_1
            next:    --
            |
            -----
            |
-->: element: -> arc_3
            next:    --
            |
            -----
            |
-->: element: -> arc_5
            next:    NULL

```

```
arc_list_4: element: -> arc_2
           next:    --
               |
           -----
           |
-->: element: -> arc_4
   next:    NULL
```

```
arc_list_6: element: -> arc_6
           next:    --
               |
           -----
           |
-->: element: -> arc_7
   next:    NULL
```

```
arc_list_5: element: -> arc_4
           next:    --
               |
           -----
           |
-->: element: -> arc_5
   next:    --
               |
           -----
           |
-->: element: -> arc_6
   next:    NULL
```

```
arc_list_7: element: -> arc_7
           next:    NULL
```

```
arc_1: name:    "put"
      direction: to
      source:   -> node_1
      sink:    -> node_2
```

```
arc_2: name:    "update"
      direction: to
      source:   -> node_1
      sink:    -> node_4
```

```
arc_3: name:    "get"
      direction: from
      source:   -> node_3
      sink:    -> node_2
```

```
arc_4: name:    "retrieve"
      direction: from
      source:   -> node_5
      sink:    -> node_4
```

```
arc_5: name:    "put"
      direction: to
      source:   -> node_5
      sink:    -> node_2
```

```
arc_6: name:    "put"
      direction: to
      source:   -> node_5
      sink:    -> node_5
```

```
arc_7: name:    "get"
      direction: from
      source:   -> node_7
      sink:    -> node_6
```

2. The Construction Data Base

This chapter demonstrates the to be built Construction Data Base. The two possibilities of constructing an application system are shown: firstly with Subsystems, secondly without Subsystems.

The entries of the Construction Data Base are grouped in a top-down manner. This eases the reading. However, the construction is performed in a bottom-up way.

2.1 The Example with Subsystems

The Activities 1, 2, and 3 are subsumed under Subsystem 1. Activity 4 is in Subsystem 2. IDA 3 therefore is a Subsystem-IDA.

AD-A137 417

THE USE OF THE MASCOT PHILOSOPHY FOR THE CONSTRUCTION
OF ADA PROGRAMS(U) ROYAL SIGNALS AND RADAR
ESTABLISHMENT MALVERN (ENGLAND) G FICKENSCHER OCT 83

22

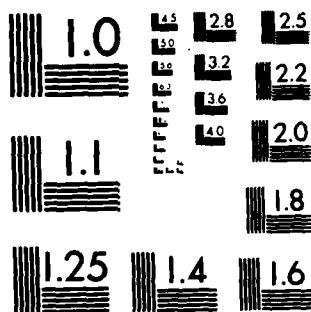
UNCLASSIFIED

RSRE-83009 DRIC-BR-90207

F/G 9/2

NL

									END				
									DATE				
									FILED				
									2 84				
									DTIC				



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

```
MASCOT_System: class:      with_subsystems
                  name:      "MASCOT_System"
                  file:      "file.MASCOT_System"
                  subsystems: -> subsystem_list
                  subsystem_IDAs: -> subsystem_IDA_list
```

```
subsystem_list: element: -> Subsystem_1
                  next:    --
                        |
```

```
-----
|
-->: element: -> Subsystem_2
      next:    NULL
```

```
subsystem_IDA_list: element: -> IDA_3
                    next:    NULL
```

```
Subsystem_1: name:      "Subsystem_1"
              file:      "file.Subsystem_1"
              parameters: -> S1_parameter_list
              activities: -> S1_activity_list
              IDAs:       -> S1_IDA_list
              subsystem_IDAs: -> S1_subsystem_IDA_list
```

```
Subsystem_2: name:      "Subsystem_2"
              file:      "file.Subsystem_2"
              parameters: NULL
              activities: -> S2_activity_list
              IDAs:       NULL
              subsystem_IDAs: -> S2_subsystem_IDA_list
```

```
S1_parameter_list: element: ----->: formal: "selector"
                  next:    NULL          actual: "50"
```

```
S1_activity_list: element: -> Activity_1    S1_IDA_list: element: -> IDA_1
                  next:    --              next:    --
                        |                  |
```

```
-----
|
-->: element: -> Activity_2
      next:    --
            |
```

```
-----
|
-->: element: -> IDA_2
      next:    NULL
```

```
-----
|
-->: element: -> Activity_3
      next:    NULL
```

S1_subsystem_IDA_list: element: -> IDA_3
next: NULL

S2_activity_list: element: -> Activity_4
next: NULL

S2_subsystem_IDA_list: element: -> IDA_3
next: NULL

Activity_1: name: "Activity_1"
root: -> Root_Template_1
parameters: NULL
access_procedures: -> A1_procs
IDAs: -> A1_IDAs

Activity_2: name: "Activity_2"
root: -> Root_Template_3
parameters: NULL
access_procedures: -> A2_procs
IDAs: -> A2_IDAs

Activity_3: name: "Activity_3"
root: -> Root_Template_2
parameters: -> A3_paras
access_procedures: -> A3_procs
IDAs: -> A3_IDAs

Activity_4: name: "Activity_4"
root: -> Root_Template_3
parameters: NULL
access_procedures: -> A4_procs
IDAs: -> A4_IDAs

A1_procs: element: ----->: formal: "write_pool"
next: -- actual: "IDA_2.update"

|

|

-->: element: ----->: formal: "write_channel"
next: NULL actual: "IDA_1.put"

A1_IDAs: element: -> IDA_2
next: --

|
-->: element: -> IDA_1
next: NULL

A2_procs: element: ----->: formal: "read_channel"
next: NULL actual: "IDA_1.get"

A2_IDAs: element: -> IDA_1
next: NULL

A3_procs: element: ----->: formal: "read_pool"
next: -- actual: "IDA_2.retrieve"

|
-->: element: ----->: formal: "write_channel_1"
next: -- actual: "IDA_1.put"

|
-->: element: ----->: formal: "write_channel_2"
next: NULL actual: "IDA_3.put"

A3_paras: element: ----->: formal: "selector"
next: NULL actual: "selector"

A3_IDAs: element: -> IDA_2
next: --

|
-->: element: -> IDA_1
next: --

|
-->: element: -> IDA_3
next: NULL

A4_procs: element: ----->: formal: "read_channel"
next: NULL actual: "IDA_3.get"

A4_IDAs: element: -> IDA_3
next: NULL

IDA_1: class: channel
 name: "IDA_1"
 file: ""
 template: -> IDA_Template_1
 data_types: -> I1_data
 parameters: -> I1_paras

IDA_3: class: channel
 name: "IDA_3"
 file: "file.IDA_3"
 template: -> IDA_Template_3
 data_types: -> I3_data
 parameters: -> I3_paras

IDA_2: class: pool
 name: "IDA_2"
 file: ""
 template: -> IDA_Template_2
 data_types: -> I2_data

I1_data: element: -> data_type_1
 next: NULL

I1_paras: element: ----->: formal: "channel data"
 next: NULL actual: "counter"

I2_data: element: -> data_type_2
 next: NULL

I3_data: element: -> data_type_1
 next: NULL

I3_paras: element: ----->: formal: "channel data"
 next: NULL actual: "counter"

data_types_list: element: data_type_1
 next: --
 |

 |
 -->: element: data_type_2
 next: NULL

data_type_1: name: "data_type_1"
 spec: "file_data_type_1.spec"
 bodie: ""

data_type_2: name: "data_type_2"
 spec: "file_data_type_2.spec"
 bodie: ""


```

IDA_template_list: element: -> IDA_Template_1
                    next:   --
                        |
                        -----
                        |
-->: element: -> IDA_Template_2
    next:   --
        |
        -----
        |
-->: element: -> IDA_Template_3
    next:   NULL

```

```

IDA_Template_1: class:      channel
                  name:      "IDA_Template_1"
                  spec:      "file.IDA_Template_1.spec"
                  bodie:     "file.IDA_Template_1.body"
                  access_procedures: -> IT1_procs
                  parameters: -> IT1_paras

```

```

IDA_Template_2: class:      pool
                  name:      "IDA_Template_2"
                  spec:      "file.IDA_Template_2.spec"
                  bodie:     "file.IDA_Template_2.body"
                  access_procedures: -> IT2_procs
                  data_types: -> IT2_data

```

```

IDA_Template_3: class:      channel
                  name:      "IDA_Template_3"
                  spec:      "file.IDA_Template_3.spec"
                  bodie:     "file.IDA_Template_3.body"
                  access_procedures: -> IT3_procs
                  parameters: -> IT3_paras

```

```

IT1_procs: element: ----->: formal: "put"
            next:   --          actual: ""
                |
                -----
                |
-->: element: ----->: formal: "get"
    next:   NULL          actual: ""

```

```

IT1_paras: element: ----->: formal: "channel_data"
            next:   NULL          actual: ""

```

```

IT2_procs: element: ----->: formal: "update"
              next:  --          actual: ""
              |
              -----
              |
-->: element: ----->: formal: "retrieve"
      next:  NULL          actual: ""

IT2_data: element: -> data_type_2
          next:  NULL

IT3_procs: element: ----->: formal: "put"
              next:  --          actual: ""
              |
              -----
              |
-->: element: ----->: formal: "get"
      next:  NULL          actual: ""

IT3_paras: element: ----->: formal: "channel_data"
          next:  NULL          actual: ""

Root_template_list: element: -> Root_Template_1
                   next:  --
                   |
                   -----
                   |
-->: element: -> Root_Template_2
      next:  --
      |
      -----
      |
-->: element: -> Root_Template_3
      next:  NULL

Root_Template_1: name:      "Root_Template_1"
                  spec:     "file.Root_Template_1.spec"
                  bodie:    "file.Root_Template_1.body"
                  parameters: NULL
                  access_procedures: -> RT1_procs
                  data_types:  -> RT1_data

Root_Template_2: name:      "Root_Template_2"
                  spec:     "file.Root_Template_2.spec"
                  bodie:    "file.Root_Template_2.body"
                  parameters: -> RT2_paras
                  access_procedures: -> RT2_procs
                  data_types:  -> RT2_data

```

```

Root_Template_3: name:          "Root_Template_3"
                  spec:          "file.Root_Template_3.spec"
                  bodie:          "file.Root_Template_3.body"
                  parameters:     NULL
                  access_procedures: -> RT3_procs
                  data_types:     -> RT3_data

```

```

RT1_procs: element: ----->: formal: "write_pool"
              next:  --          actual: ""
              |
              -----
              |
-->: element: ----->: formal: "write_channel"
    next:  NULL          actual: ""

```

```

RT1_data: element: -> data_type_1
           next:  --
           |
           -----
           |
-->: element: -> data_type_2
    next:  NULL

```

```

RT2_paras: element: ----->: formal: "selector"
           next:  NULL          actual: ""

```

```

RT2_procs: element: ----->: formal: "read_pool"
              next:  --          actual: ""
              |
              -----
              |
-->: element: ----->: formal: "write_channel_1"
    next:  --          actual: ""
    |
    -----
    |
-->: element: ----->: formal: "write_channel_2"
    next:  NULL          actual: ""

```

```

RT2_data: element: -> data_type_1
           next:  --
           |
           -----
           |
-->: element: -> data_type_2
    next:  NULL

```

```

RT3_procs: element: ----->: formal: "read_channel"
           next:  NULL          actual: ""

```

```
RT3_data: element: -> data_type_1
          next:     NULL
```

2.2 The Example without Subsystems

This Subchapter shows the head of the Construction Data Base, if Subsystems are not concerned. The other entries of the data base are already shown by Subchapter 2.1. The only exception is the entry "A3_Paras".

```
MASCOT_System: class:    without_subsystems
                 name:    "MASCOT_System"
                 file:    "file.MASCOT_System"
                 activities: -> activity_list
                 IDAs:     -> IDA_list
```

```
activity_list: element: -> Activity_1
                next:     --
                |
                -----
                |
                -->: element: -> Activity_2
                next:     --
                |
                -----
                |
                -->: element: -> Activity_3
                next:     --
                |
                -----
                |
                -->: element: -> Activity_4
                next:     NULL
```

```
A3_paras: element: ----->: formal: "selector"
          next:     NULL          actual: "50"
```

```
IDA_list: element: -> IDA_1
          next:     --
          |
          -----
          |
          -->: element: -> IDA_2
          next:     --
          |
          -----
          |
          -->: element: -> IDA_3
          next:     NULL
```

3. Data Types of the IDAs

Two data type packages are used in the example. The first one represents the types used by IDA 1 and 3, the second one the types used by IDA 2, and, of course, by the respective Activities.

```
PACKAGE data_type_1 IS
    SUBTYPE counter IS integer RANGE 0 .. 49;
END data_type_1;

PACKAGE data_type_2 IS
    TYPE store_counter IS
        RECORD
            number : integer;
            text   : string(1 .. 100);
        END RECORD;
    SUBTYPE key IS integer RANGE 1 .. 100;
END data_type_2;
```

4. The IDA Templates

There are three templates (one for every IDA) in the example. This approach is chosen to show that Channel can share the same data type definition.

4.1 IDA Template 1

IDA Template 1 implements a Channel which can store up to one hundred data objects of a non-abstract data type temporarily. Putting an object to the Channel is impossible, if the data area is full. Getting an object from the Channel is impossible, if the data area is empty. The data objects are read in the order in which they are put to the Channel. Readers and writers have equal priority in accessing the Channel.

```
GENERIC
    TYPE channel_data IS PRIVATE;

PACKAGE IDA_Template_1 IS
    PROCEDURE put(x : IN channel_data);
    PROCEDURE get(x : OUT channel_data);
END IDA_Template_1;
```

```

PACKAGE BODY IDA_Template_1 IS

    size      : CONSTANT integer := 100;
    data_area : ARRAY (1 .. size) OF channel_data;
    count     : integer RANGE 0 .. size := 0;
    in_index  ,
    out_index : integer RANGE 1 .. size := 1;

    TASK t IS
        ENTRY read (x : OUT channel_data);
        ENTRY write(x : IN  channel_data);
    END t;

    TASK BODY t IS
    BEGIN
        LOOP
            SELECT
                WHEN count < size =>
                    ACCEPT write(x : IN channel_data) DO
                        data_area(in_index) := x;
                    END;
                    in_index := in_index MOD size + 1;
                    count := count + 1;
                OR
                WHEN count > 0 =>
                    ACCEPT read(x : OUT channel_data) DO
                        x := data_area(out_index);
                    END;
                    out_index := out_index MOD size + 1;
                    count := count - 1;
            END SELECT;
        END LOOP;
    END t;

    PROCEDURE put(x : IN channel_data) IS
    BEGIN
        t.write(x);
    END put;

    PROCEDURE get(x : OUT channel_data) IS
    BEGIN
        t.read(x);
    END get;

END IDA_Template_1;

```

4.2 IDA Template 2

IDA Template 2 implements a Pool. The Pool has a data area with as many entries as the range of the type "key" denotes. All entries of the data area are preset with an initial value. After reading an entry it is reset to the initial value. The key which prescribes the storage place of an object in the data area is derived from the object itself. Readers have precedence over writers. Writers overwrite already stored data objects.

```
WITH data_type_2;  
USE data_type_2;
```

```
GENERIC
```

```
PACKAGE IDA_Template_2 IS
```

```
    PROCEDURE update (x : IN store_counter ; y : OUT key);  
    PROCEDURE retrieve(x : IN key ; y : OUT store_counter);
```

```
END IDA_Template_2;
```

```
PACKAGE BODY IDA_Template_2 IS
```

```
    data_area : ARRAY (1 .. key'last) OF store_counter :=  
        (1 .. key'last =>  
            store_counter'(number => 0,  
                text => (1..store_counter.text'last) => ' '));
```

```
    TASK t IS
```

```
        ENTRY write(x : IN store_counter ; y : OUT key);  
        ENTRY read (x : IN key ; y : OUT store_counter);  
    END t;
```

```
    TASK BODY t IS
```

```
    BEGIN
```

```
        LOOP
```

```
            SELECT
```

```
                ACCEPT read(x : IN key ; y : OUT store_counter) DO  
                    y := data_area(x);  
                    data_area(x) :=  
                        store_counter'(number => 0,  
                            text => (1..store_counter.text'last) => ' ');
```

```
            END;
```

```
        OR
```

```
            WHEN read'count = 0 =>  
                ACCEPT write(x : IN store_counter ; y : OUT key) DO  
                    y := x.number MOD key'last + 1;  
                    data_area(y) := x;
```

```
            END;
```

```
        END SELECT;
```

```
    END LOOP;
```

```
END t;
```

```
PROCEDURE update(x : IN store_counter ; y : OUT key) IS  
BEGIN
```

```
    t.write(x , y);  
END update;
```

```
PROCEDURE retrieve(x : IN key ; y : OUT store_counter) IS  
BEGIN
```

```
    t.read(x , y);  
END retrieve;
```

```
END IDA_Template_2;
```

4.3 IDA_Template_3

IDA Template 3 implements a Channel. The Channel can only hold exactly one object temporarily. Therefore reading and writing must follow each other in a strict order, starting with writing. The data objects which are passed through the Channel can be of any non-abstract data type.

GENERIC

```
TYPE channel_data IS PRIVATE;

PACKAGE IDA_Template_3 IS

    PROCEDURE put(x : IN channel_data);
    PROCEDURE get(x : OUT channel_data);

END IDA_Template_3;

PACKAGE BODY IDA_Template_3 IS

    data_area : channel_data;

    TASK t IS
        ENTRY read (x : OUT channel_data);
        ENTRY write(x : IN channel_data);
    END t;

    TASK BODY t IS
    BEGIN
        LOOP
            ACCEPT write(x : IN channel_data) DO
                data_area := x;
            END;
            ACCEPT read(x : OUT channel_data) DO
                x := data_area;
            END;
        END LOOP;
    END t;

    PROCEDURE put(x : IN channel_data) IS
    BEGIN
        t.write(x);
    END put;

    PROCEDURE get(x : OUT channel_data) IS
    BEGIN
        t.read(x);
    END get;

END IDA_Template_3;
```


5. The Root Templates

Three Root templates are used by the example. Activity 2 and 4 are derived from the same template.

5.1 Root Template 1

An Activity derived from Root Template 1 communicates with other Activities through the formal procedures "write_pool" and "write_channel". In this example the procedures indicate accesses to IDA_1 which is a Channel and to IDA_2 which is a Pool but this is not prescribed. Every IDA one of whose access procedures matches one of the formal procedures can be used as communication link.

Root Template 1 generates an object called "pool_object". Depending on the result of a random generator which delivers either "true" or "false" the procedures "write_pool" or "write_channel" are selected as communication links respectively.

```
WITH data_type_1 , data_type_2;  
USE data_type_1 , data_type_2;
```

```
GENERIC
```

```
  WITH PROCEDURE write_pool(x : IN store_counter ; y : OUT key);  
  WITH PROCEDURE write_channel(x : IN counter);  
PROCEDURE Root_Template_1;
```

```
PROCEDURE Root_Template_1 IS
```

```
  pool_object    : store_counter;  
  pool_key       : key;  
  channel_object : counter;
```

```
  -- other declarations including the boolean function "random"
```

```
BEGIN
```

```
  -- list of statements including the begin of a loop  
  -- and the generation of "pool_object"
```

```
  IF random  
  THEN  
    write_pool(pool_object , pool_key);  
  ELSE  
    channel_object := pool_object.number MOD (counter'last + 1);  
    write_channel(channel_object);  
  END IF;
```

```
  -- list of statements including the end of the above mentioned loop
```

```
END Root_Template_1;
```

5.2 Root Template 2

Root Template 2 communicates through three access procedures. It reads using the procedure "read_pool" from an IDA depending on the value of a randomly generated key "pool_key". The template selects either the procedure "write_channel_1" or the procedure "write_channel_2" for passing messages depending on its value parameter and on the value of the read object "pool_object".

```
WITH data_type_1 , data_type_2;
USE  data_type_1 , data_type_2;

GENERIC
  WITH PROCEDURE read_pool(x : IN key ; y : OUT store_counter);
  WITH PROCEDURE write_channel_1(x : IN counter);
  WITH PROCEDURE write_channel_2(x : IN counter);
PROCEDURE Root_Template_2(selector : IN integer);

PROCEDURE Root_Template_2(selector : IN integer) IS

  pool_object      : store_counter;
  pool_key         : key;
  channel_object   : counter;

  -- other declarations

BEGIN

  -- list of statements including the randomly generation of "pool_key"
  -- and the begin of a loop

  read_pool(pool_key , pool_object);

  IF pool_object.number > selector AND selector <= (counter'last + 1)
  THEN
    channel_object := pool_object.number MOD selector;
    write_channel_2(channel_object);
  ELSE
    channel_object := pool_object.number MOD (counter'last + 1);
    write_channel_1(channel_object);
  END IF;

  -- list of statements including the end of the above mentioned loop

END Root_Template_2;
```

5.3 Root Template 3

Root Template 3 only reads messages using the procedure "read_channel" and consumes them.

```

WITH data_type_1;
USE data_type_1;

GENERIC
  WITH PROCEDURE read_channel(x : OUT counter);
PROCEDURE Root_Template_3;

PROCEDURE Root_Template_3 IS
  channel_object : counter;

  -- other declarations

BEGIN
  -- list of statements including the begin of a loop
  read_channel(channel_object);

  -- list of statements including the end of the above mentioned loop

END Root_Template_3;

```

6. Subsystem-IDAs

One Subsystem-IDA is considered by the example. IDA_3 is derived from IDA Template 3 using a type of the package "data_type_1" as definition for its formal data type "channel_data".

```

WITH data_type_1;
USE data_type_1;

PACKAGE IDA_3 IS
  NEW IDA_Template_3(channel_data => counter);

```

7. Subsystems

The Activities are subsumed under two Subsystems. Activity 1, 2, and 3 form Subsystem 1, Activity 4 Subsystem 2. Subsystem 1 has a value parameter to supply Activity 3 with a proper value for its value parameter.

```

WITH data_type_1 , data_type_2;
WITH IDA_Template_1 , IDA_Template_2;
WITH Root_Template_1 , Root_Template_2 , Root_Template_3;
WITH IDA_3;
USE data_type_1 , data_type_2;
USE IDA_Template_1 , IDA_Template_2;
USE Root_Template_1 , Root_Template_2 , Root_Template_3;
USE IDA_3;

PROCEDURE Subsystem_1(selector : IN integer)
IS

    PACKAGE IDA_1 IS
        NEW IDA_Template_1(channel_data => counter);

    PACKAGE IDA_2 IS
        NEW IDA_Template_2;

    TASK Activity_1;

    TASK BODY Activity_1 IS
        PROCEDURE Activity_1_Root IS
            NEW Root_Template_1(write_pool      => IDA_2.update ,
                                write_channel => IDA_1.put);
        BEGIN
            Activity_1_Root;
        END Activity_1;

    TASK Activity_2;

    TASK BODY Activity_2 IS
        PROCEDURE Activity_2_Root IS
            NEW Root_Template_3(read_channel => IDA_1.get);
        BEGIN
            Activity_2_Root;
        END Activity_2;

    TASK Activity_3;

    TASK BODY Activity_3 IS
        PROCEDURE Activity_3_Root IS
            NEW Root_Template_2(read_pool      => IDA_2.retrieve ,
                                write_channel_1 => IDA_1.put ,
                                write_channel_2 => IDA_3.put);
        BEGIN
            Activity_3_Root(selector);
        END Activity_3;

BEGIN

    NULL;

END Subsystem_1;

```

```

WITH Root_Template_3;
WITH IDA_3;
USE Root_Template_3;
USE IDA_3;

PROCEDURE Subsystem_2
IS

    TASK Activity_4;

    TASK BODY Activity_4 IS
        PROCEDURE Activity_4_Root IS
            NEW Root_Template_3(read_channel => IDA_3.get);
        BEGIN
            Activity_4_Root;
        END Activity_4;

BEGIN

    NULL;

END Subsystem_2;

```

8. The Main Program with Subsystems

The main program consists of two tasks which call the two Subsystems respectively. Subsystem 1 is supplied with a proper actual value parameter.

```
WITH Subsystem_1 , Subsystem_2;
```

```
PROCEDURE MASCOT_System  
IS
```

```
    TASK Subsystem_1_task;
```

```
    TASK BODY Subsystem_1_task IS  
    BEGIN
```

```
        Subsystem_1(50);  
    END Subsystem_1_task;
```

```
    TASK Subsystem_2_task;
```

```
    TASK BODY Subsystem_2_task IS  
    BEGIN
```

```
        Subsystem_2;  
    END Subsystem_2_task;
```

```
BEGIN
```

```
    NULL;
```

```
END MASCOT_System;
```

9. The Main Program without Subsystems

If no Subsystems are considered, the main program is formed like a Subsystem. There is a task for every Activity. Every IDA must be created by instantiating the proper template. The root procedures must be supplied with proper actual value parameters, if requested (for example, look at Activity 3).

```

WITH data_type_1 , data_type_2;
WITH IDA_Template_1 , IDA_Template_2 , IDA_Template_3;
WITH Root_Template_1 , Root_Template_2 , Root_Template_3;
USE data_type_1 , data_type_2;
USE IDA_Template_1 , IDA_Template_2 , IDA_Template_3;
USE Root_Template_1 , Root_Template_2 , Root_Template_3;

PROCEDURE MASCOT_System IS

    PACKAGE IDA_1 IS
        NEW IDA_Template_1(channel_data => counter);

    PACKAGE IDA_2 IS
        NEW IDA_Template_2;

    PACKAGE IDA_3 IS
        NEW IDA_Template_3(channel_data => counter);

    TASK Activity_1;
    TASK BODY Activity_1 IS
        PROCEDURE Activity_1_Root IS
            NEW Root_Template_1(write_pool    => IDA_2.update ,
                                write_channel => IDA_1.put);

        BEGIN
            Activity_1_Root;
        END Activity_1;

    TASK Activity_2;
    TASK BODY Activity_2 IS
        PROCEDURE Activity_2_Root IS
            NEW Root_Template_3(read_channel => IDA_1.get);

        BEGIN
            Activity_2_Root;
        END Activity_2;

    TASK Activity_3;
    TASK BODY Activity_3 IS
        PROCEDURE Activity_3_Root IS
            NEW Root_Template_2(read_pool      => IDA_2.retrieve ,
                                write_channel_1 => IDA_1.put ,
                                write_channel_2 => IDA_3.put);

        BEGIN
            Activity_3_Root(50);
        END Activity_3;

    TASK Activity_4;
    TASK BODY Activity_4 IS
        PROCEDURE Activity_4_Root IS
            NEW Root_Template_3(read_channel => IDA_3.get);

        BEGIN
            Activity_4_Root;
        END Activity_4;

    BEGIN
        NULL;
    END MASCOT_System;

```

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. ORIC Reference (if known)	2. Originator's Reference Report 83009	3. Agency Reference	4. Report Security Classification unclassified	
5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location Royal Signals and Radar Establishment			
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location			
7. Title The Use of the MASCOT philosophy for the construction of Ada programs				
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 Fickenscher, G	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp. ref.
11. Contract Number	12. Period	13. Project	14. Other Reference	
15. Distribution statement Unlimited				
Descriptors (or keywords)				
continue on separate piece of paper				
Abstract The development of computer based systems poses major problems on the people involved. Both, MASCOT (the official design methodology of the UK Ministry of Defence for real-time systems) and Ada (the official programming language of the US Department of Defence for embedded computer systems) claim to offer a solution to the majority of these problems. MASCOT is a programming support environment which is independent of a particular programming language, but it defines its own runtime kernel for parallel execution of different program parts. Ada, on the other hand, offers language constructs to express parallelism of program parts, but Ada enforces a particular design methodology with its language rules.				

Continued Summary

This paper investigates whether it is feasible to combine the MASCOT methodology with the programming language Ada. It demonstrates a possible implementation of a MASCOT Construction Data Base in Ada, and it explains the combination of MASCOT and Ada by using a simple example.

