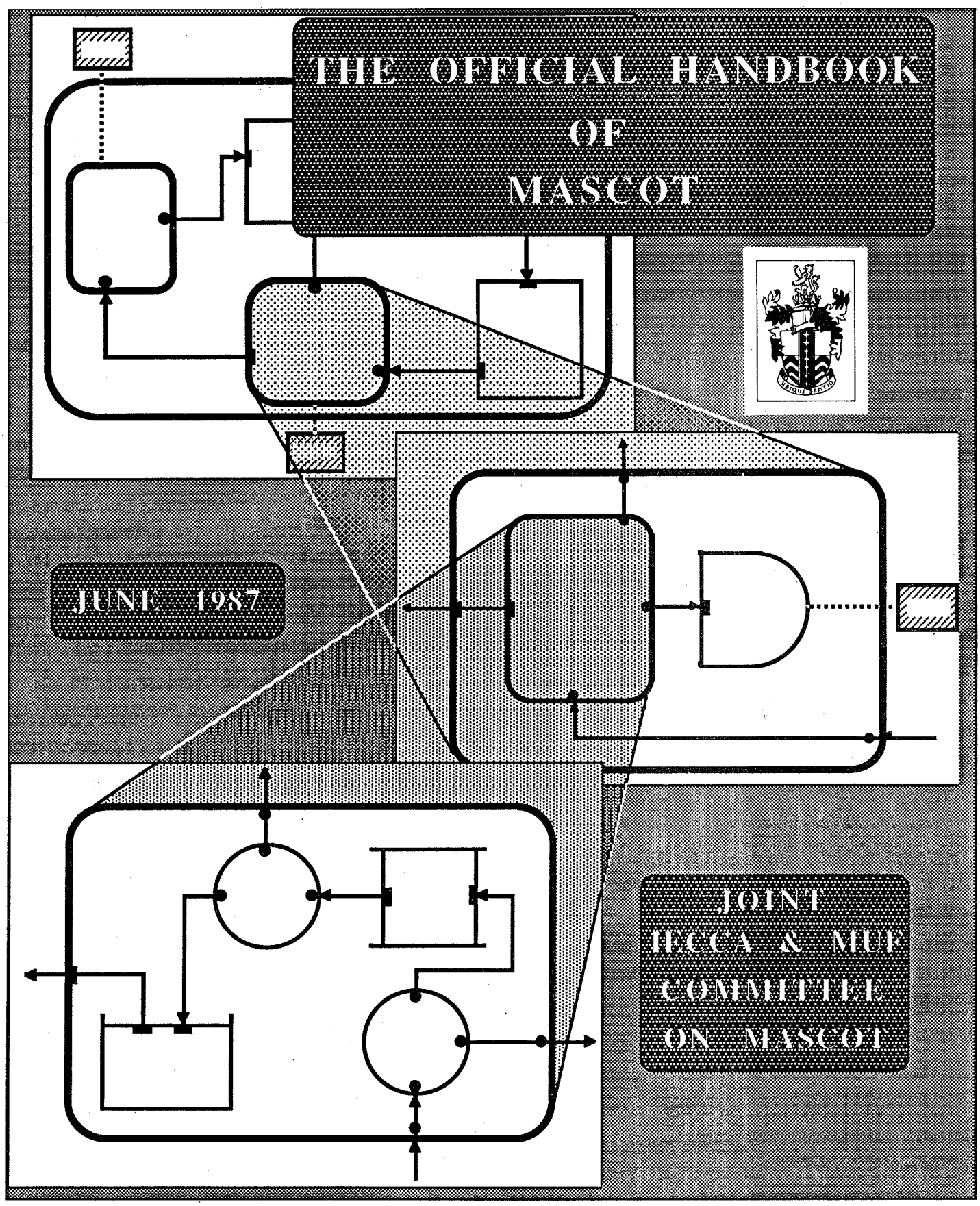

THE OFFICIAL HANDBOOK
OF
MASCOT
JUNE 1987
JOINT
IEE.CCA & MUF
COMMITTEE
ON MASCOT

THE OFFICIAL HANDBOOK OF MASCOT

VERSION 3.1 ISSUE 1

JUNE 1987

This document is issued by the Joint IECCA and MUF Committee on Mascot (JIMCOM). Amendments and additional copies are issued by:

Computing Division
N Building
Royal Signals and Radar Establishment
St Andrews Road
Malvern
Worcestershire
WR14 3PS

CONTENTS

AUTHOR'S PREFACE

> ***'Lastly, I stand ready with a pencil in one hand, and a sponge in the other, to add, alter, insert, expunge, enlarge and delete, according to better information. And if these my pains shall be found worthy to pass a second impression, my faults I will confess with shame, and amend with thankfulness to such as will contribute clearer intelligence unto me'***
>
> **Preface to 'The History of the Worthies of England'**
>
> **Thomas Fuller (1662)**

Such, as technical author of the Official Handbook of Mascot Version 3.1, has been my attitude over the past two and a half years to the scientific worthies of the Mascot 3 Definition Team; albeit my pencil and sponge are of an electronic variety. Some sections of the handbook, having stimulated particular controversy among the Team or having been more than normally misunderstood by me or having been especially savaged by the pre-publication reviewers, have been 'found worthy' of a whole long series of impressions. But it has all been well worth the effort from my point of view. When, in 1984, I commenced work as a freelance lecturer and writer, I could hardly have expected the good fortune of becoming involved in such a stimulating and rewarding project. I am grateful to all concerned.

I would like to express my thanks, first of all, to the Royal Signals and Radar Establishment for financing the task of writing the Handbook. My gratitude is specially due to Ken Hayter and Keith Oliver who shared the job of Technical Authority for the project. I would also like to thank the Royal Military College of Science, and particularly Tony Sammes as head of the Computing Science Group, for performing project administration. To the members of JIMCOM I am greatly obliged for their faith in ratifying my original appointment as author and for their forbearance in accepting a series of revised target completion dates which seemed at times to be diverging to infinity. I would like to thank all those who, in the course of the limited public review of the draft Handbook, provided helpful comments leading to clearer exposition and pointed out many typographic, grammatic and orthographic errors.

Finally, to the members of the Definition Team itself and especially to Hugo Simpson, Ken Jackson, Tony Riddiough and Bill Taylor, I owe an immense debt of gratitude. Throughout my period of involvement in the work, they have striven to communicate their ideas to me and to correct my misconceptions with unfailing patience. While rightly insisting that the concepts of Mascot 3 be presented to the world with technical accuracy and appropriate relative emphasis, they have allowed me to take full responsibility in matters of presentation and have tolerated my occasional stylistic idiosyncrasy with commendable resignation.

George Bate

Wantage, May 1987

PREFACE

The development of complex, computer based systems poses major problems to the people involved. These problems encompass both managerial aspects, concerned with control of the overall development, and technical aspects concerned with the interaction of the individually designed components of the system. Mascot offers a wide-ranging and homogeneous approach to the development of such systems. It provides significant contributions to the solution of both managerial and technical problems.

Historical Background

Mascot was originated by Ken Jackson and Hugo Simpson over the period 1971 to 1975. After the initial implementation work was completed, the Royal Signals and Radar Establishment (RSRE) formed the Mascot Suppliers Association (MSA) in order to effect the transfer of Mascot technology into industry. The MSA, which consisted of individuals from several companies and MOD establishments, produced, in 1978, an 'Official Definition of Mascot'. This document described what came to be known, retrospectively, as Mascot 1 and provided a definitive reference for implementors and teachers of Mascot while ideas and methods continued to evolve.

In 1980 a sub-committee of the MSA, drawing its membership from the following:

Admiralty Surface Weapons Establishment
Royal Military College of Science
Royal Signals and Radar Establishment
Computer Analysts and Programmers (Reading) Ltd
Ferranti Computer Systems Ltd
GEC Computers Ltd
Software Sciences Ltd
Systems Designers Ltd

drafted a much more comprehensive presentation of the Mascot concepts as 'The Official Handbook of Mascot'. This handbook, which was reissued in 1983, constitutes the standard reference for Mascot 2 and has received an extensive distribution. A companion volume to the 1983 issue, 'Additional Features to Integrate Mascot with Coral 66', provides a formal syntactic description of a set of extensions to the MOD standard programming language which make it a suitable vehicle for Mascot applications. This language was named AF Coral 2.

The drafting of these documents was one of the last actions of the MSA before it was disbanded, having achieved its major objectives. Responsibility for maintaining the Mascot standard, in so far as it is intended

for use in government projects, was taken over by the Inter Establishment Committee on Computer Applications (IECCA). A joint committee of the MOD and the DoI, IECCA is composed wholly of official representatives. In order that the liaison with industry and the computing community generally, so successfully initiated by the MSA, could be maintained and extended, another organisation, the Mascot Users' Forum (MUF), was set up in 1980. Informal symposia, open to all actual and potential Mascot users, suppliers and supporters, are arranged by the MUF and some 80 official, industrial and academic bodies have been represented.

To provide a convenient basis for the continued technical development of Mascot, IECCA and the MUF formed, in 1981, the Joint IECCA and MUF Committee on Mascot (JIMCOM). It is under the aegis of JIMCOM that the work on Mascot 3, the subject of this present version of the Official Handbook, has been carried out. This new Mascot definition has been developed by a team in which the following have been principal contributors:

Lawrence Collingbourne (Systems Designers plc)
Gerry Docherty (YARD Ltd)
Giles Forster (MOD -EQC)
Ken Jackson (Systems Designers plc)
Tony Riddiough (Software Sciences Ltd)
Hugo Simpson (British Aerospace plc)
Bill Taylor (Ferranti Computer Systems Ltd)

The enormous contribution of George Bate who was the technical author responsible for translating the ideas from the development team into a consistent, coherent text is gratefully acknowledged. Acknowledgement is also due for the support of RSRE. Finally the contribution of the people who commented on the draft versions of the Handbook is gratefully acknowledged.

Handbook Organisation and Conventions

This Handbook has been written principally for the benefit of users and potential users of Mascot. The presentation is therefore broadly tutorial. However, within this general approach an attempt has been made to be as helpful as possible both to the implementors of Mascot and to those concerned with assessing and evaluating the resulting implementations. There are three major sections. The first of these is introductory, providing the background to the present stage of Mascot development and presenting, in an informal manner, the main innovations of Mascot 3. Then follows the Official Definition which is the essential core of the book. It contains both descriptive passages suitable for those requiring an overall understanding of the ideas and rather more formal material intended to be used for reference purposes. Finally, there is a section devoted to guidance in the use of Mascot. It is of course only through practical experience that the optimum application of the Mascot 3 features will emerge but the advice given here reflects the rationale upon which they have been devised.

The new concepts which this handbook introduces into the Mascot philosophy demand an extended technical vocabulary for their description. Devising acceptable and consistent terminology has not proved easy. There are a limited number of possible words available (if we reject the idea of coining entirely new ones) and they are all inseparable from their existing associations in both technical and everyday usage. The importance of the glossary which appears at the end of the handbook can, therefore, hardly be emphasised too strongly. It contains definitions of all the Mascot technical terms. In order that the reader will be aware that a word is being employed in a precise sense, all such instances are signalled throughout the Definition by the use of **bold type**. This will help to make more comprehensible those passages in which technical terms have had to be used before being fully explained in the text. It will be advisable, even for those already familiar with earlier versions of Mascot, to consult the glossary regularly while reading through the handbook for the first time.

The only other typographical convention is the use of ***bold italic*** for text which would otherwise need to be in quotation marks. Examples include identifiers used in sample program fragments and names invented for the syntactic elements of the design representation language.

Mascot designs have two parallel forms of representation: graphical and textual. The former presents no problem here. Its conventions are well defined and are summarised in an appendix. There is no barrier to their standard use in all Mascot applications. The textual form, however, does raise difficulties. The Mascot tradition of programming language independence is retained in Mascot 3 even though, for many, the choice in the past has been 'any language provided it is Coral' and in the future will presumably be 'any language as long as it is Ada*'. Neither of these languages is ideal as a vehicle for expressing the textual form of the Mascot design representation though either will serve in practical use.

The solution adopted has been to invent a design representation language to fulfil a twofold purpose. First, it serves here to define and explain the design constructs in a rigorous and consistent manner and can be used for a similar purpose in future publications. Second, it is proposed as the notation in terms of which practical Mascot 3 designs will actually be devised and communicated. While it is very desirable that some automatic means of translation, such as a pre-processor, should be made available for mapping these designs into particular programming languages, there is no implication that this is an essential prerequisite to the use of Mascot. Experienced programmers have long recognised that the language 'in which they program' need not be the (more or less inadequate) implementation language which has to be used for other, often non-technical, reasons. The same considerations apply here and the sole criterion must be, as in the past, that the concepts described in the Mascot Definition are capable of being expressed in the chosen implementation language.

The design representation language is broadly Pascal-like in that, where a suitable Pascal convention exists, it has been adopted. This choice was made partly on the grounds that Pascal is widely familiar to the international computing community and partly because of its use as the base language in the

development of the NATO preferred programming language, Ada*. The syntax of the language itself has been defined by means of the type of syntax diagrams first employed by Wirth to describe Pascal. The author's experience of teaching programming languages at a variety of levels has produced a strong conviction that such diagrams provide the best available means of combining rigour with comprehensibility. A complete set is presented, for ease of reference, in appendix A which also presents the syntax in BNF together with an index to the Handbook itself. Where differences occur between the syntax diagrams used in the text and the corresponding diagrams in Appendix A, those in the appendix constitute the full definition.

* Ada is a registered trademark of the U.S. Government - Ada Joint Program Office

CHAPTER 1

INTRODUCTION

1. INTRODUCTION

Mascot is a Modular Approach to Software Construction Operation and Test which incorporates:

a) a means of design representation
b) a method for deriving the design
c) a way of constructing software so that it is consistent with the design
d) a means of executing the constructed software so that the design structure remains visible at run time
e) facilities for testing the software in terms of the design structure.

Of particular importance in the design representation is the ability to represent, directly, concurrent functions and the data flows between them. An equally important facet of the method is the fact that individual components in the design structure are de-coupled from each other. This has a significant impact on both the design method and the testing strategy and leads directly to a form of 'component technology' familiar in all other branches of engineering. It causes a design to be expressed as a structure (or assembly) of interconnected components each of which is of a specific type. Thus each component type has its own characteristics and embodies constraints on where and how it may be connected to other component types. But, and this is a critical feature, no component refers directly to another component. Such interconnection information is specified in a separate form rather like an engineering drawing.

The software structure, by its insistence upon decoupling, also has a significant impact upon the components' potential for re-use and specifically makes the creation of test systems very much more straightforward.

Mascot can be and has been used in a wide range of application areas. It is however aimed primarily at real-time embedded application areas where the software is complex and highly interactive.

1.1 Structure, Modularity and Management

One important motivation for the provision of modularity stems from the need to employ a team of people on one single job. In such circumstances it is essential that each member of the team can be allocated a job to do which contributes to the overall success of the project. Many modularity schemes have been devised to address this problem. Most of them involve creating an overall design and then carving up the design into chunks which can be allocated to an individual team member. A key feature of Mascot is that, because the design is expressed as an interconnected network of otherwise independent components, each component can be developed in isolation from the others. Then the developed components can

be brought together into assemblies or sub-assemblies at a later stage in the secure knowledge that, because the interconnection constraints have been policed during development, the components will fit together and will have a high probability of working together correctly.

Thus, through a modularity scheme based on sound and secure component technology, Mascot provides not only the basic ingredients for sound team management, but also allows the use of traditional approaches to engineering management.

1.2 Derivation of Mascot

1.2.1 Historical (see also the Preface to this Handbook)

The antecedents of Mascot can be traced back to the period 1965 -1970. During this time the originators of Mascot were involved respectively with the development of control programs for an automatic computer controlled radar and a multi-access operating system for the control of on-line experiments in real time, and the in-service maintenance of various operational, embedded, real-time RAF systems. This early experience in tackling the problems of embedded real-time systems culminated in the successful development of a large air defence system. It was during this last project that the originators came together and began to investigate the possibility of creating an alternative and well defined method of software development. These investigations led to Mascot.

1.2.2 Technical

Much of the motivation behind Mascot lay in the apparent preoccupation with control-flow design which was prevalent in the late 60s. For complex real-time systems it was obvious that there could not be a unique control flow design which satisfied all the functional requirements because of the highly stochastic nature of the inputs. Therefore the originators looked to their training as electrical engineers for inspiration. Here they discovered that concurrency was a key feature together with the notion of data flow or information flow from one stage to another. For example, in a radio receiver RF signals are amplified by the RF stage and the output is detected by the demodulator stage. The demodulator output is fed to an audio amplifier and its output is fed to the loudspeaker. Thus there is the notion of information flowing from one (concurrent) stage to the next and the very important concept of well defined interfaces between stages to decouple the design of one stage from the next. There is no notion of control flow because each stage is performing its own special function all the time. Thus this model of analogue electrical circuits became the foundation for the Mascot design representation and method of software construction.

From the outset, the originators held the view that writing sequential programs (or modules) was not a problem - or rather it was not a problem which Mascot would address! Instead Mascot would concentrate on what was perceived to be a more difficult problem, namely that of representing concurrency and data flow in embedded real-time systems.

1.3 Software Structure in Mascot

1.3.1 General

One of the key problems with software is its intangibility and this is especially true of large systems. Hence, a way of describing the software is required which presents just sufficient detail for the purpose. This usually involves a top-down presentation so that an overall view can be given at first. Further, more detailed, views follow until the lowest level of detail, which is usually the source text in an appropriate programming language, is reached. The key requirement of a design representation medium is this ability to present a design at appropriate levels of detail.

In Mascot the starting point was to represent the design graphically with emphasis on the presentation of data flow and concurrency. From such a diagram one can gain an overall impression of what each component has to do and how the overall functionality of the software system is achieved. Also from the diagram it is possible to write down an equivalent textual representation of the whole system structure and of the individual components required to build that structure. Then the other details of the components can be added as the design effort proceeds. Once components have been built, test systems can be created in which the components can be tested. Finally the complete design structure can be built for system testing.

These characteristics are evident in both Mascot 2 and Mascot 3. In the following sections we describe first the Mascot 2 method and then indicate how and why Mascot 3 differs. A fuller introduction to Mascot 3 design representation can be found in section 2.1. Here we merely touch upon the salient features of both Mascot 2 and Mascot 3 in order to make the comparison and to identify the motiviation for updating Mascot 2.

1.3.2 Mascot 2

The basic notion in Mascot is that the flow of data through a system, from input sensors to output actuators, is controlled solely by a set of concurrent software processes. These processes are known as 'activities' and are separately scheduled by a run-time system usually referred to as a Mascot 'kernel'. Data enter and leave the system through 'devices' which are software accessible registers in the hardware with which the software system communicates. The data are moved around the system and transformed by activities. Mascot activities thus need to co-operate with each other by passing data but they are not allowed to communicate directly; their immediate communication is with special modules, provided for this purpose, called 'Intercommunication Data Areas' and usually referred to as IDAs. IDAs are passive components which exist only to satisfy the intercommunication requirements of activities. They contain data areas which are completely private to the IDA and support the intercommunication requirements by providing a procedural interface which can be used by activities. Thus the designer can design in terms of concurrent processes which are purely sequential (ie activities) and IDAs which are passive but encapsulate the interactions between the activities.

This represents a significant separation of concerns:

- activities are sequential processes concerned primarily with perfoming a single function and communicating with a (usually small) set of IDAs

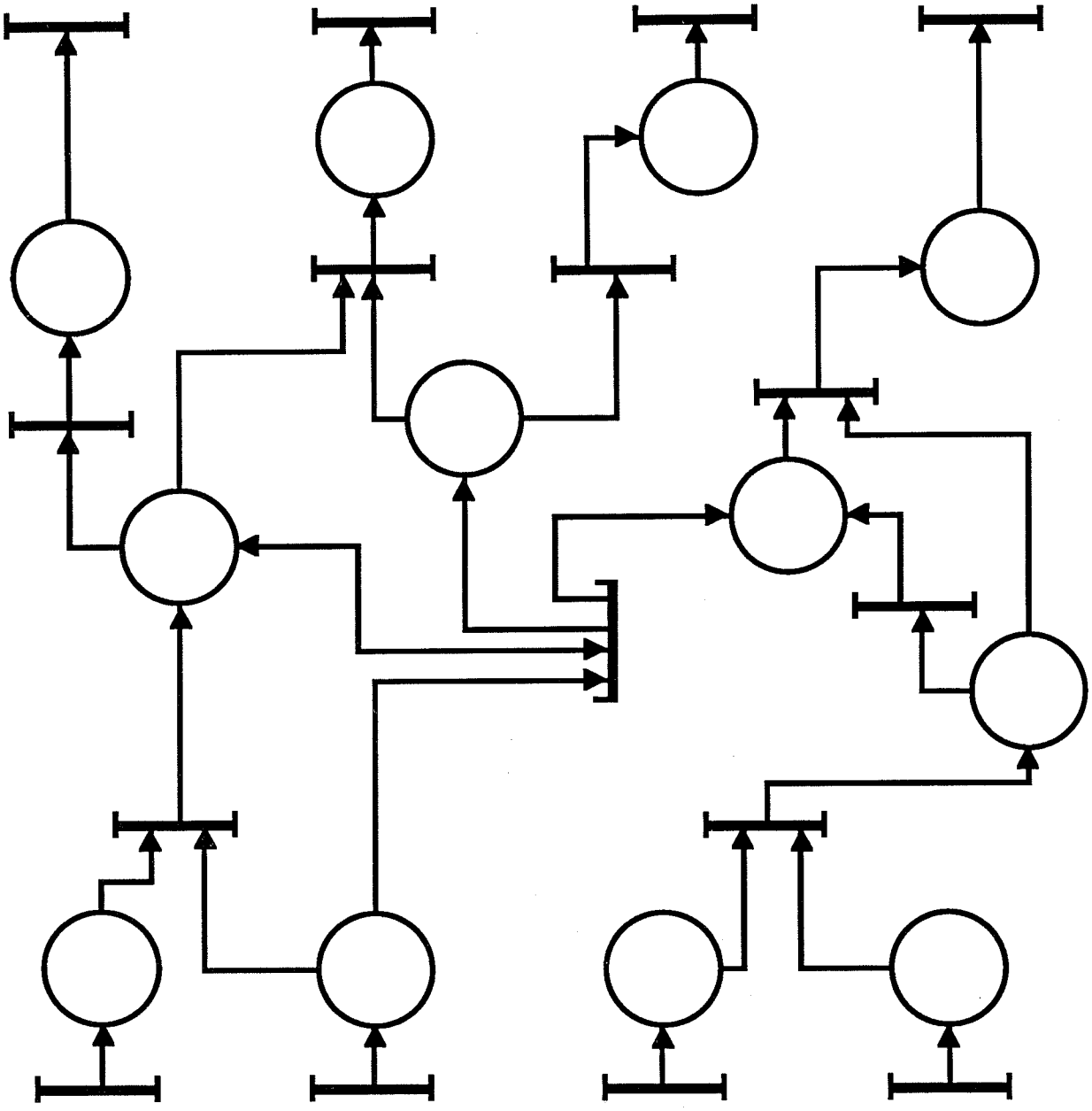
- IDAs contain any data sharable by activities. Access to the data is provided by a procedural interface whose constituent procedures must ensure the integrity of the data in the IDA by using the synchronisation facilities of the Mascot kernel.

Two distinct classes of IDA have been identified in Mascot: the 'channel' and the 'pool'. The channel is used to pass data between activities on a 'producer/consumer' basis. Producer activities send messages via an input interface and consumer activities receive messages via an output interface. There can be a buffer of messages in a channel at any instant. These are messages which have been sent but not yet received (ie they are currently in transit). This property of a channel is useful in relieving the consumer activity of the necessity of running in synchronism with the producer activities.

The pool is used to hold data which may need to be referred to by activities and, in particular, in cases where the frequency or pattern of references to the data is completely independent of the frequency or pattern of operations which bring it up to date. Thus there could be many references without any change being made to the data or the data might be changed many times between successive references.

It is important to realise that Mascot does not provide a specific and fixed set of IDAs. Facilities are provided for a designer to define and build the specific types of IDAs required by his application.

The graphical representation used for Mascot 2 is known as the ACP (Activity, Channel, Pool) Diagram. An example is given below together with a key to the symbols used. Note that the producer/consumer characteristic of channels is indicated by connecting one side of the symbol to producer activities and connecting the other side to the consumer activity. The direction of data flow is indicated by the arrow heads.




ACTIVITY
CHANNEL
POOL
DEVICE

The ACP diagram depicts the instances of the activities, channels and pools required. The types of the components can also be depicted on the ACP diagram (usually by enclosing the type in parentheses).

As mentioned earlier activities do not refer directly to IDA instances. Instead they are defined and coded in terms of the set of IDA types with which they need to communicate. During the software construction process the activities are created(ie manufactured) and then 'connected' (i.e. assembled) to the set of IDA instances required by the design and expressed in the ACP diagram. The unit of construction in Mascot 2 is the 'subsystem'. This is merely a collection of activities which have been connected to their IDAs at the same time. The subsystem is also the unit which can be controlled at run time by being started, terminated, halted or resumed. This provides the facility for evolving a Mascot network. The operational network consists of the set of subsystems which have been 'formed' and 'started'. By forming new subsystems, terminating old subsystems and starting the new subsystems it is possible to change an operational system as time passes.

1.3.3 Mascot 3

Motivation for the development of Mascot 3 has stemmed from two considerations. First, experience of Mascot 2 in practical use has inevitably revealed, within its proper province of application, some areas of weakness which it is desirable to remedy. Second, the emerging prospect of implementing systems for large, multiprocessor networks has brought with it a need for more powerful and flexible means of design expression than those available in Mascot 2. The aim has been, therefore, both to consolidate and to extend the Mascot method.

Perhaps the most significant refinement which Mascot 3 brings to existing Mascot concepts concerns the manner of expressing network connections. Mascot 2 networks are formed from system elements which have been created from templates held in a constructional database. Every activity template contains a fixed number of external connections each of which is expressed as a reference to an IDA template. An assembly of inert activities is converted into a network of communicating activities by supplying IDAs of the appropriate type, created that is from appropriate templates, to store information and control its transmission in each of the data flow paths. The choice of valid network configurations is thus constrained by a type checking mechanism based on the fact that activity templates contain direct references to IDA templates. Such an arrangement allows much greater flexibilty than one in which the references are to specific IDA instances.

Although this Mascot 2 style of expressing design definition is very flexible it does contain one major restriction, namely that an activity template depends upon IDA templates. In Mascot 3 this dependency has been changed so that the inter-dependency between activities and IDAs is focussed on the interface which exists between them. This interface exists implicitly in Mascot 2 and consists of the set of access procedures implemented in the IDA. In Mascot 3 the procedural interface is specified explicitly and is

called an 'access interface'. Thus in Mascot 3 an activity template is specified in terms of its requirement to use one or more access interfaces. A component derived from that template may be connected to a set of IDA components (derived from their templates) subject to the condition that the access interfaces provided by the IDA are those required by the activity.

The terminology introduced into Mascot 3 to describe these ideas is that an IDA 'provides' an access interface at a 'window' and an activity 'requires' an access interface at a 'port'. Thus activities can have several ports, as indeed they could in Mascot 2. However, in Mascot 3, IDAs can provide several windows whereas in Mascot 2 an IDA could provide only a single window.

This greater degree of design decoupling provides several advantages in terms of design expression and eliminates some of the causes of inefficiency in Mascot 2 design. First, an activity may be connected to any IDA type provided that the port to window connectivity constraint is satisfied (ie that the access interface required at the port is that provided at the window). This means, for example, that for test purposes an activity can be connected to an IDA which is of a different type from the one to which it is connected in an operational network.

Second, the ability of an IDA to provide more than one window means that, in Mascot 3, a channel can provide two interfaces, one for the writers and one for the readers. The same idea can be extended to pools where a set of access interfaces can be provided each encompassing a limited capability corresponding to the particular requirements of the activities using it. The use of multiple windows by the designer is also valuable in identifying how the functions of a system are distributed in relation to the IDA's position in the network.

A third major advantage concerns the freedom, in Mascot 3, to design IDAs which provide several windows of the same type. Thus a special significance can be attached, within the IDA, to any of these windows. For example, one window might be given priority over the others or actions within the IDA can be made dependent on the location of the caller within the connected network. The ability of an IDA to provide several windows, combined with the decoupling arising from separate definition of the access interface, is considered to be one of the key contributions of Mascot 3.

Another major departure from previous Mascot philosophy is the adoption of hierarchical structure. A Mascot 2 design is conceived in terms of a flat, data flow network whose nodes consist of alternate data processing and data communication elements. While higher level descriptions may be used in the process of devising the network, they have no permanent standing and are not recognised as design entities by the supporting constructional database. The final, essentially two-dimensional structure may be partitioned into an arbitrary patchwork of subsidiary networks, the adjacent members of which share common communication elements. These are the Mascot 2 subsystems which subsequently constitute units of control at run time.

In Mascot 3, the role of the subsystem has been greatly elevated. While it continues to represent a subsidiary network, it no longer possesses shared components but may be used as a network node in its own right. As such it is capable of being connected to either processing or communication elements and so may perform the function of either or behave as a mixture of the two. It may also possess lower level subsystems among its components so as to facilitate an hierarchical form of design expression. At the same time, with appropriate database support, it makes it possible to develop and capture a design, progressively.

The modules of text which represent subsystems (and systems) in the Mascot 3 database contain, between them, all the information needed to establish the content and connectivity of the complete network. The processing of these modules, supported by all the remaining modules which define the system, leads to the construction of a template from which a complete collection of application software can be created and loaded into appropriate locations in the target hardware. They thus embody all the information which, in the Mascot 2 method of software construction, is provided at the stages of system element creation and subsystem formation.

Whereas in Mascot 2, therefore, the connectivity of the software network is not established until the last stage of construction, in Mascot 3 it is established as the first stage. The advantage of the Mascot 2 approach is that it readily supports evolutionary construction, the ability to make adjustments to the network without regenerating the system or even terminating its execution. This may be less easy to achieve in Mascot 3 but a compensatory gain, in addition to hierarchical design expression, is the facility of validating the overall design structure in terms of the inter-module dependencies before any detailed implementation coding has been submitted to the database.

It will be clear from the earlier discussion that a Mascot 3 subsystem is a composite entity. It defines a set of interconnected components. Most other design elements have composite as well as simple forms. For example, a composite IDA defines a network of internal, component IDAs. This is made possible by another extension of the Mascot philosophy which allows IDAs to communicate with each other directly without an intervening active element. Such relatively complex constructions are particularly useful where the IDA is required to control communication between activities located in different parts of distributed hardware.

The fundamental notion in Mascot 2 that activities must only communicate via IDAs is retained in Mascot 3. However, because Mascot 3 IDAs can have ports, it is possible that the interaction between any two activities may involve more than one IDA.

The concept of composite templates in Mascot 3 extends to sequential as well as network decomposition. An individual thread of execution, an activity, may be composed of a number of separately created components which communicate with each other through well defined procedural interfaces. These components share with the simple form of activity the ability to make external network

connections. Since the components may themselves be composite the facilities exist for hierarchical expression of this level of the design also.

The Mascot model for device handling is essentially unaltered in Mascot 3. It is, however, catered for in a more formal manner by the provision of a new class of design element, called a server, which is dedicated to communication with hardware devices. It is similar to the generalised, Mascot 3 form of IDA but is the only design element which communicates with devices and to facilitate this is allowed to contain an interrupt handler.

1.4 Method

In Mascot 2 the method could be divided into three phases:

a) Network Design - which created the ACP diagram and identified the purpose of each component
b) Component Design - in which each component was designed and coded
c) Integration and Test - in which each component was tested first individually and then in conjunction with other components

In Mascot 3 the same set of phases can be identified but there are some additional tasks within each:

a) Network Design in Mascot 3 is an iterative process involving (potentially) the creation of an hierarchy of subsystems, IDAs and servers.
b) Component Design in Mascot 3 can involve the decomposition of activities into lower level components
c) Integration and Test in Mascot 3 is similar to Mascot 2 except that it must take account of the additional levels of decomposition.

1.5 Development Environment

The Mascot 3 development environment is far less specific than that defined for Mascot 2 . This is primarily because Mascot 3 is considered to be more capable of being used as a stand alone design method than Mascot 2. An important contributory factor here is the need to work with languages such as Ada which do not allow the intimate integration that has been possible with, say, Coral 66. Therefore, the Mascot 3 development environment has been defined in terms of a set of functions to control the progressive capture of a design. These functions are known as the 'status progression commands' and allow a template in Mascot 3 to be first 'registered', then 'introduced', and finally 'enrolled'. These operations work primarily on one specific template, but they also require a specific status to be achieved by other templates upon which that template depends.

1.6 Execution Environment

The execution environment of Mascot 3 is again very much less specific than for Mascot 2. None of the facilities specified have been significantly changed, but, in recognition of the existence of languages which directly support concurrency, the Mascot run-time facilities (ie those provided by the Mascot kernel) are no longer mandatory.

CHAPTER 2

DESIGN REPRESENTATION

2.1 AN INFORMAL INTRODUCTION

2.1.1 Introduction

This part of the Official Definition of Mascot, 'Design Representation', contains the quintessence of the Mascot approach. It is the portion of the Handbook to which those familiar with the earlier (1981) edition will wish to give closest attention in order to gain an understanding of the ideas which have been developed during the past five years. An attempt has been made to present these new concepts with the rigour and completeness befitting a formal definition while, at the same time, making it as easy to read as possible. Opinions will differ as to how successfully these objectives have been attained but it is a safe assumption that most readers will find some of the material relatively demanding. Hence the need for an informal introduction.

The exposition in this section is neither rigorous nor complete and should not be taken as definitive. While it is, of course, accurate as far as it goes, it is in no sense a substitute for the sections which follow but rather is intended to establish a framework within which the detail, presented later, may more readily be understood. It concentrates on the simpler aspects of each topic in order to introduce the principal concepts and terms. The Definition describes a set of facilities judged sufficiently powerful, in their entirety, for use in addressing the design of extremely complex computer systems. Here, however, the more complex constructions and most of the supplementary features are omitted in the interests of displaying the essential simplicity of the underlying ideas. In practice, the users of Mascot will adopt as many of its features as may be required for the application in hand.

2.1.2 Design Representation

The architecture of Mascot designs is expressible in two equivalent forms: graphical and textual. Each one may readily be derived from the other. For example, a design which is conceived and developed in the graphical form may be transformed, in a wholly mechanical manner, into the textual form and hence progressively captured in the **Mascot database** to establish the structure of the software. Implementation is then completed by specifying details of the interfaces through which the components of the system communicate, together with the data types with which they are concerned, and by supplying the executable code expressed in whatever implementation language has been adopted. Alternatively, a system might be designed directly in the textual notation and the graphical form subsequently derived from it to become the central feature of its design documentation and to provide a primary medium of discussion for everyone involved with the system throughout its lifecycle.

One of the prime features of the Mascot method is concurrency. A typical design defines, in an hierarchical manner, a set of parallel co-operating processes. At the higher levels of the hierarchy these parallel threads of execution are bunched together in constructional units called **subsystems**. Progressive expansion of the **subsystems** separates the larger bunches into smaller ones and eventually, at the lower levels, teases out the individual threads. These individual units of concurrency in

a Mascot design are known as **activities**. They are executed in a standard run-time environment provided by a collection of **context software** whose functions are implicitly available to the application software. The interface between the **context** and the application software is expressed in a form which is generally compatible with the style of the application software modules and is known as the **context interface**.

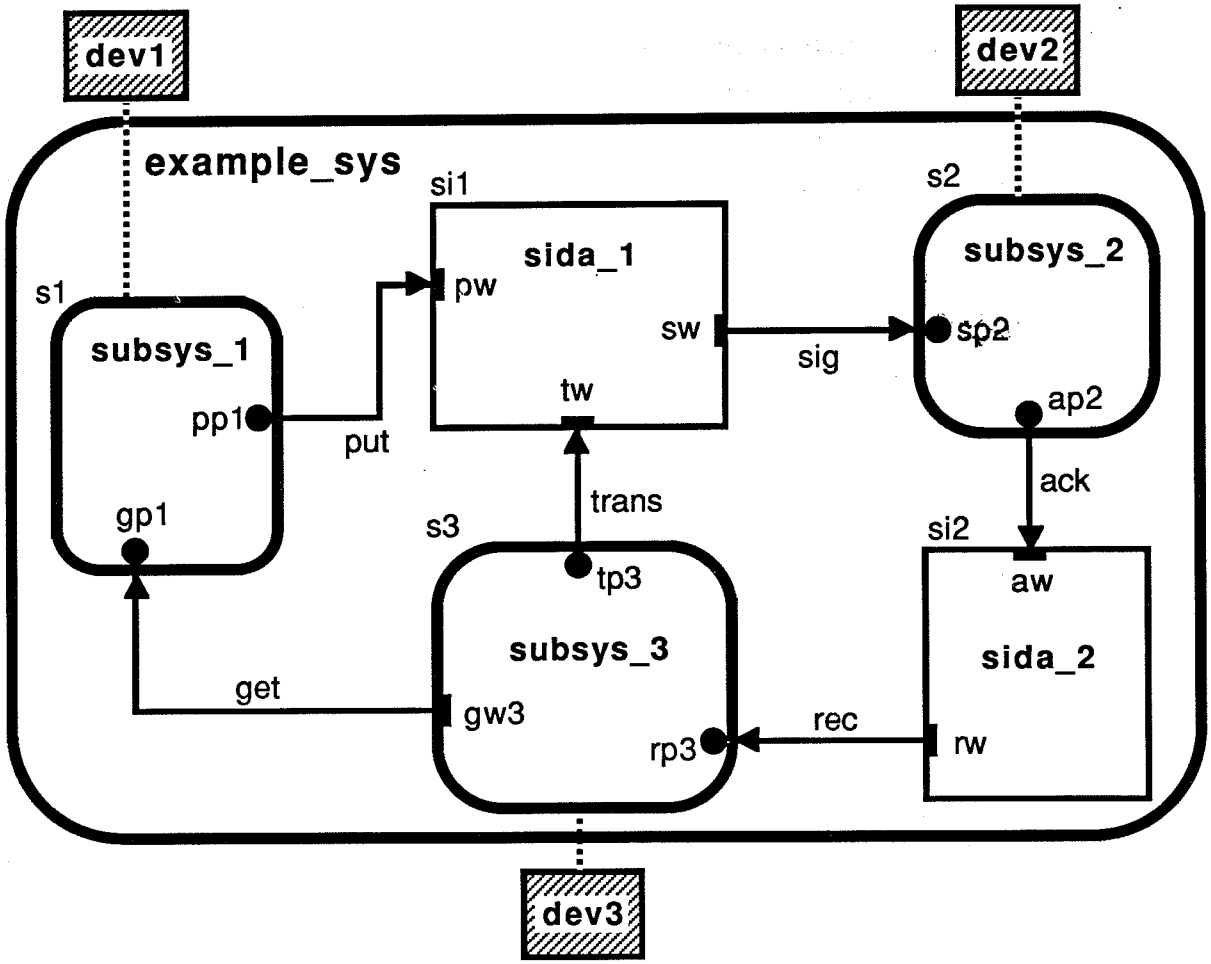
The hierarchical nature of this structure permits the design to be viewed at various levels of abstraction, examination of any one of which immediately highlights the second salient feature of the Mascot design representation. This is data flow. Each level of the design is conceived as a network through which data is transmitted from one active entity (**subsystem** or **activity**) to another. The ultimate sources and sinks of this information are provided by a set of hardware devices which are regarded as being outside the Mascot **system** but with which communication may take place through a class of software design elements called **servers** which are dedicated to this purpose.

2.1.3 A Sample Outline Design

In Section 5.1 of the Handbook the approach to the development of a Mascot design is described in detail. For our purposes here we will suppose that a design has already been completed to the point at which the overall software structure has been established. The diagram is drawn and there exists in the **Mascot database** a **module**, that is a named textual representation, for each of the design elements that has been used. Furthermore, all the inter **module** references have been checked and found valid.

We are not concerned with what our imaginary **system** is designed to do. Identifiers have deliberately been chosen for their inherent lack of meaning, or else to be so general as to achieve the same effect, in order that there shall be no temptation to be distracted by this question. This would, of course, be as reprehensible in a real design as failing to use meaningful identifiers in a sequential program. In practice it is recommended that **template** names reflect the function provided by the **template**; wheras **component** names should reflect the purpose of the **component** in the network which contains it. The only purpose of the **system** we are about to examine, however, is to demonstrate aspects of Mascot design representation. It should not, in particular, be taken as exemplifying specially recommended practice.

The natural place to begin is with the diagram representing the top level of the design.



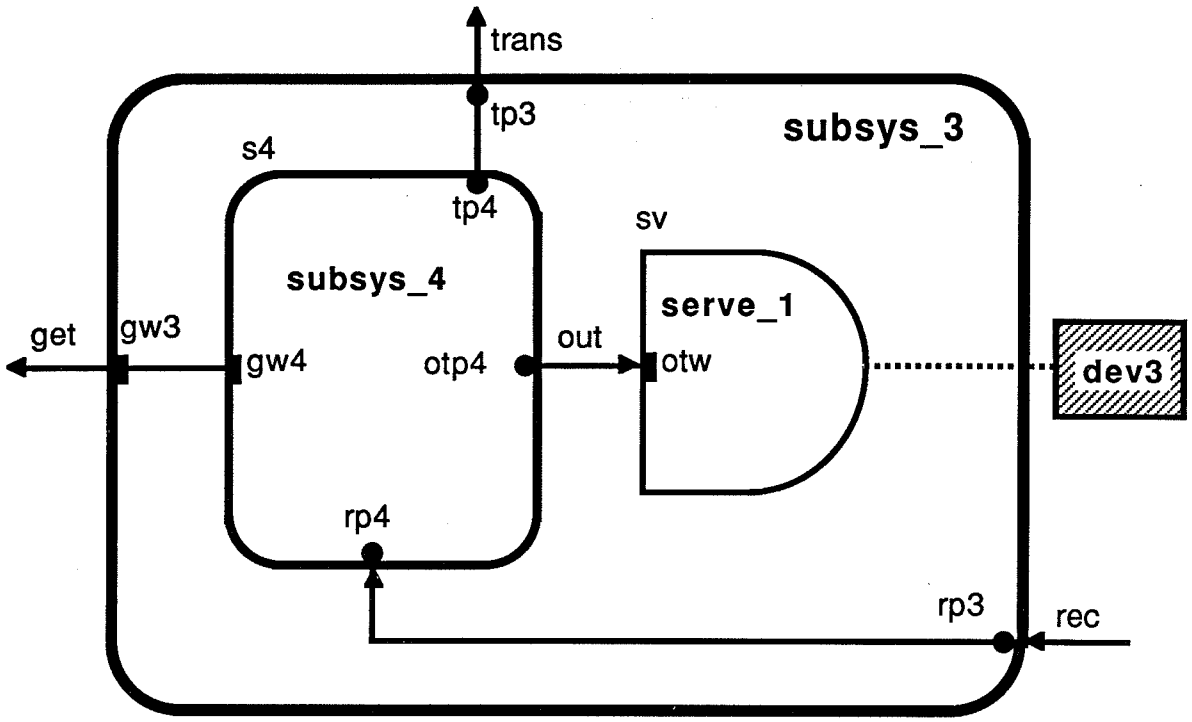
This diagram shows our example **system**. It corresponds to what, in the Mascot method, is called the 'initial design response'. It shows, on the outside of the **system**, the set of hardware devices the **system** is required to interact with. It shows, on the inside, a set of high level design components which represent the initial design of the **system**. The **system** itself is symbolised by the round cornered rectangle which identifies the boundary between hardware and software and indicates the flow of data into, out of and within the **system** in very broad brush terms. The name of this particular **system** is ***example_sys***. It consists of five communicating **components** three of which, like the **system** itself, are symbolised by round cornered boundaries and represent **subsystems**. Throughout the Mascot graphical convention round corners generally indicate active entities and, although occasional exceptions can occur, it is normally safe to assume that a **subsystem** contains at least one of the concurrent threads of execution which constitute the active constituents of the **system**. The three **components**, ***s1***, ***s2*** and ***s3*** may therefore be thought of as being executed in parallel.

The two remaining **components** of **system** ***example_sys*** illustrate the feature which, more than any other, distinguishes Mascot from other approaches to the problems of large scale concurrency. In order that asynchronously executed processes may exchange information in a secure manner, it is necessary to provide mechanisms to effect mutual exclusion and cross-stimulation for use at the points where data is transferred to or from common storage areas. As explained inSection 4.2 of the Handbook, failure to do this may lead to information becoming corrupt and failure to do it adequately may result in the processes becoming deadlocked. In many approaches to the organisation of parallel, co-operating processes, these

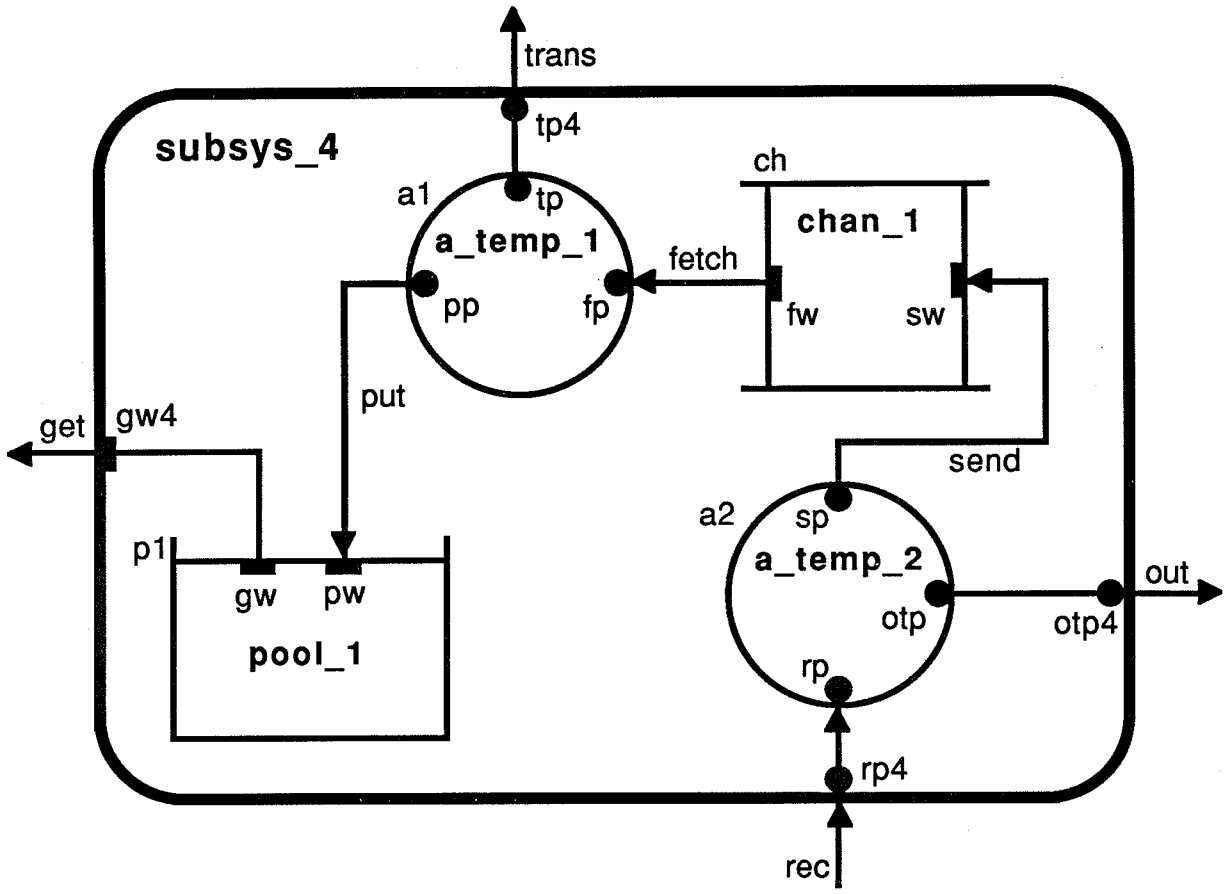
fundamental mechanisms are not made directly available by the run-time system. Instead, a set of higher level operations/facilities such as monitors, message passing or rendezvous are provided. In Mascot the view was taken that, in order to obtain the optimum performance which is always vital in in embedded systems, it was best for the run-time system to make the low level facilities directly available. This has the advantage of making the run-time system small and efficient. It also gives the designer freedom to design exactly the right set of higher level operations required for his application. The design entity in which all these requirements are satisfied is called an **Intercommunication data area** or, more succinctly, an **IDA**. It is the responsibility of **IDA** designers to implement the required operations as **access procedures** (or functions) within an **IDA**. These operations use the low level synchronisation facilities to maintain both data flow and data integrity. A further contribution is made to system integrity by ensuring that only **IDA**s contain data which can be the subject of interaction from several **acivities** and that only **IDA**s may use the low level synchronisation facilities. The most general form of **IDA** is represented graphically by a rectangle; ***si1*** and ***si2*** are examples.

The thin lines, bearing arrow-heads, which link the **subsystem** and **IDA** symbols into a network are lines of data flow known in Mascot as **paths**. Thus, data flows from **subsystem** ***s2*** to **subsystem** ***s3*** along the two **paths**, labelled ***ack*** and ***rec*** respectively, which enter and leave the **IDA** ***si2***. In **IDA** ***si1*** a merging of information flow occurs with **paths** ***put*** and ***trans***, from ***s1*** and ***s3***, entering and **path** ***sig*** to ***s2*** leaving. It will be seen that in one instance a **path**, ***get***, links two **subsystems** directly. However, as we shall discover, this does not reflect any failure to carry out the necessary synchronisation. The concepts of both **paths** and **IDA**s will be discussed in greater detail later when our example system has been expanded to reveal the lower levels of its structure.

We shall eventually return to this **system** diagram and examine the **module** (textual unit) which is equivalent to it. For the moment, leaving some of its detail unexplained, we shall consider what further information is needed for software construction. Obviously it is necessary to know how to create the **components**. A pattern, or in Mascot terms a **template**, is required for each of the three **subsystems** and two **IDA**s. Consider, for example, the **component** labelled ***s3***. The identifier ***subsys_3*** which appears inside the corresponding symbol is the name of the **template** from which this **subsystem** is created. Expansion to a further level of decomposition reveals the graphical representation of its internal composition.



It will be seen that the **template**'s external connections match those of the **component** which it describes. Data flows into the **subsystem** along a **path** labelled ***rec*** and out along **paths** labelled ***get*** and ***trans***, respectively. All three of these **paths** are connected, internally, to one of the **template**'s two **components**, ***s4***, which is immediately recognisable as another **subsystem**. The second **component**, called ***sv*** and represented by a D-shaped symbol, is an example of a **server**. This is the design element, referred to earlier, which is able to communicate with external, hardware devices. A device is represented here by a hatched rectangle joined to the **server** by a broken line. We shall return to this diagram later but, for the present, further discussion will once again be postponed in favour of performing one more level of decomposition. In order to show what is required for the creation of **component** ***s4***, it is expanded to reveal its **template**, ***subsys_4***.



No further network decomposition is possible in this branch of the hierarchy. The **components** of **subsystem** ***subsys_4*** include two individual **activities**, represented by the two large circular symbols labelled ***a1*** and ***a2***. **Activities**, as already indicated, are fundamental processing **elements**. Each one is to be regarded as implementing a separate parallel thread. Any further analysis of them can only be in terms of sequential, rather than network, decomposition.

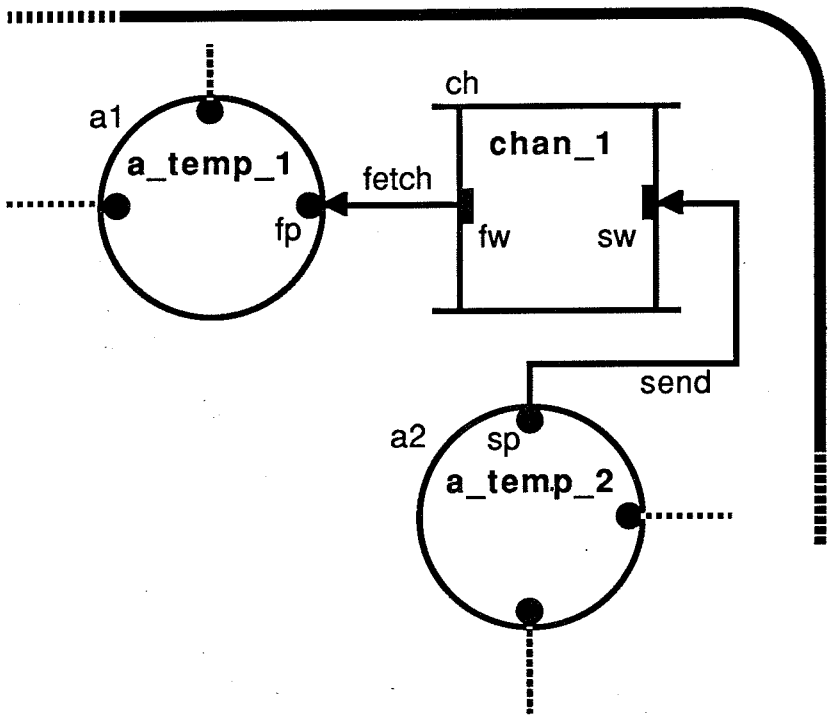
The **template**, ***subsys_4*** also contains two **IDAs**. They are represented by slightly modified versions of the simple rectangular symbol seen earlier in the **system** diagram. This shows them to be special cases corresponding to the **channels** and **pools** of previous versions of Mascot. The **channel** is characterised by a destructive read operation; data flowing through it is temporarily accommodated in internal storage which may become full as a result of repeated write operations or empty as a result of repeated read operations. In a **pool** it is the write operation which is destructive. Its contents consist of a collection of variables which are given initial values when the system commences execution and which may subsequently be examined and updated.

2.1.4 The Communication Model

Having now looked briefly at the graphical representation of these three hierarchically related levels of the imaginary design, we shall now consider each in more detail. For this purpose it will be convenient to proceed from the lowest level upwards, examining the **modules** which represent the various **templates** as we go. But first it is necessary to deal with a topic which is so fundamental to Mascot as to

demand separate treatment. So far, data flow through a **network** has been taken for granted. It is now time to examine the Mascot communication model in more detail.

The fundamental concepts are illustrated in the diagram below which shows a simplified fragment of the **subsystem** ***subsys_4***.



Here we have the simple case of a producer **activity**, ***a2***, supplying data to a consumer **activity**, ***a1***. The **IDA** (a **channel**), connected between the two **activities** and represented by a modified rectangular symbol labelled ***ch***, acts as a temporary repository for items of data en route from ***a2*** to ***a1***. An intermediate storage buffer, together with coding to operate on it, is encapsulated in the **IDA**. ***ch*** might, for example, contain a procedure which adds an item of data to the buffer and a procedure which removes items from the buffer. It is in the coding of such procedures, known as **access procedures**, that the mechanisms for effecting cross-stimulation and mutual exclusion are employed.

Procedures encapsulated by **IDAs** are made selectively available for use by **activities** through the concept of **windows**. These are represented graphically by the small, filled rectangles labelled ***sw*** and ***fw*** which appear, each at the end of a **path**, just inside the boundary of the **IDA** symbol. A **window** of an **IDA** makes externally available a sub-set of the interactions which the **IDA** is able to provide. The nature of the interactions provided at a particular **window** matches the type of the **path** connected to it. This is indicated on the diagram as an identifier labelling the **path**. Thus ***sw*** and ***fw*** are connected to **paths** of type ***send*** and ***fetch***, respectively.

The type of a **path** is defined in a **module** called an **access interface**. This is classified as a **specification** as distinct from the **templates** which define the types of Mascot components such as **activities**, **IDAs**, **servers**, **subsystems** and **systems**. It contains sufficient information to allow the

corresponding set of interactions to be invoked. Typically this includes procedure headings and, indirectly, definitions of the data types which appear in their parameter lists. The types of the two labelled **paths** in our **subsystem** fragment, for example, might be defined as follows:

```
ACCESS INTERFACE send ;
   WITH flow_data ;
   PROCEDURE insert ( item : flow_data ) ;
END .

ACCESS INTERFACE fetch ;
   WITH flow_data ;
   PROCEDURE extract ( VAR item : flow_data ) ;
END .
```

The **WITH** clause which appears in each of these **modules** is a reference to the common source from which they obtain their definition of the data-type ***flow_data*** needed in both of the **access procedures**. This is provided by a **specification** known as a **definition** which might, in this instance, take the following form:

```
DEFINITION flow_data ;
   TYPE
      flow_data = RECORD

                  END;
END .
```

Definitions are the means by which other Mascot **modules**, whether representing **specifications** or **templates**, share data-type definitions. Depending on the particular programming language being employed, Mascot implementations may impose additional rules concerning the naming of **definitions** and the point in the progressive elaboration of a design at which they are required to be present in the **database**.

Coding capable of implementing procedures ***insert*** and ***extract*** is included in the **template**, ***chan_1***, which defines the **IDA** ***ch***. In outline, ***chan_1*** looks like this:

```
CHANNEL chan_1 ;
   PROVIDES sw : send ;
            fw : fetch ;
   .
   ACCESS PROCEDURE insert ( item : flow_data ) ;
   .
   END ;
   ACCESS PROCEDURE remove ( VAR item : flow_data ) ;
   .
   END ;
   fw.extract = remove
END .
```

After the heading, which names the **template**, a **PROVIDES** section lists all the **windows** of the **IDA**, giving each a name and a type which relates it to an **access interface**. The procedures which implement the interactions specified in the **Interfaces** are identified in the body of the **IDA** by the

language word **ACCESS**. Other procedures might be declared in the **template** together with data structures such as the storage buffer and its associated pointers. These program entities would all be local to **IDAs** (such as ***ch***) created from the **template** and inaccessible to all other components.

This example demonstrates the two ways in which the correspondence between the **access procedures** and the **window** specifications may be established. In the case of procedure ***insert*** the correspondence is established implicitly by name. Procedure ***remove***, on the other hand, is explicitly identified with the **access interface** procedure specified as ***extract***. This is achieved through an access equivalence list at the end of the **module**. Thus while simple cases can be dealt with simply, an unrestricted facility exists whereby internally defined procedures may be allocated between the **windows** of the **IDA**.

Returning to the fragmentary **subsystem** diagram, it will be seen that each of the two **paths** that we have been discussing connect, at the ends remote from the **IDA windows**, to small, filled circles labelled ***sp*** and ***fp***, respectively. These are situated just inside the boundaries of the two **activity** symbols and are known as **ports.** They are the means of expressing the requirement of an **activity** for the interactions specified in an **access interface**. For a valid network connection to be established between a **port** of one component and a **window** of another, they must refer to the same **access interface**. Appropriate **ports** are specified in the **activity templates** ***a_temp_2*** and ***a_temp _1*** as follows:

```
ACTIVITY a_temp_2 ;
   REQUIRES sp : send ;

END .


ACTIVITY a_temp_1 ;
   REQUIRES fp : fetch ;

END .
```

Thus **objects**, such as ***a2***, created from ***a_temp_2***, contain coding to invoke the interactions specified in **access interface** ***send.*** The **port** name is used as a selector:

```
VAR
   val : flow_data ;
BEGIN

   sp.insert( val ) ;
```

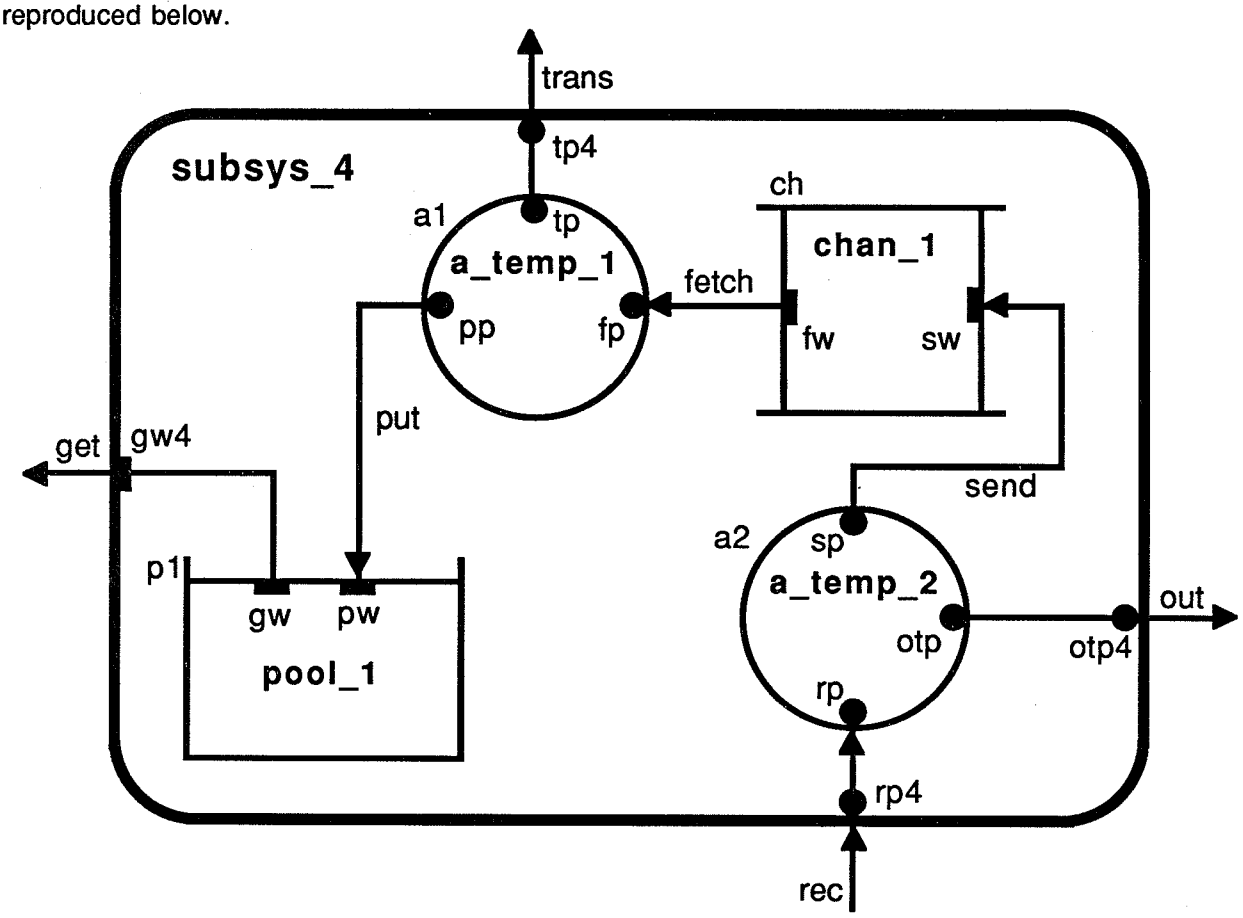
With the interconnections as described, **activity component** ***a2*** would, by this means, invoke execution of procedure ***insert*** in the **IDA component** ***ch.*** Similarly **activity** ***a1*** would invoke procedure ***remove*** (recall the access equivalence list) of ***ch*** by:

```
VAR
   next : flow_data ;
BEGIN

   fp.extract( next ) ;
```

Furthermore, the ***flow_data*** items can be transmitted, in this way, from one **activity** to the other via a buffer in **IDA** ***ch*** which is not directly accessible to either.

2.1.5 A Subsystem Containing Activities and IDAs

These, then, are the principal features of the Mascot communication model. All the internal features of **template** ***subsys_4*** should now be understandable from the diagram which, for convenience, is reproduced below.



In addition to the interactions just described in detail, ***a1*** also transfers information into the **pool** ***p1*** along a **path** of type ***put***. Each of these internal **paths** has a **port** at its **activity** end and a **window** at its **IDA** end. Notice that in this example data in some **paths** flows from a **port** to a **window** and in others from a **window** to a **port**. Data flow in both directions along the same **path** is also possible.

All the remaining connections on this diagram pass through the **subsystem** boundary. They represent the external dependencies of **subsystems** created from this **template**. As we have seen, the external dependencies of **activities** and **IDAs** are expressed as **ports** and **windows**. The same is true of

subsystems which may possess both **ports** and **windows**. This one, for example, has three **ports** and one **window** as may be seen more clearly in the higher level diagram, representing the **template** ***subsys_3***, of which it is a **component**.

All the coding of a **subsystem template** is encapsulated in its **components** and, consequently, each of the **template's ports** or **window** must be connected directly to a **port** or **window** of its **components**. Indeed, it is reasonable to regard the specification of a **subsystem port** or **window** as a method of making a **port** or **window** of one of its **components** available for connection outside the **subsystem**. This is illustrated on a diagram by **'port** to **port'** and **'window** to **window'** connections. Thus the **window** ***gw4***, of type ***get***, on the boundary of **subsystem** ***subsys_4*** is equated to the **window** ***gw*** of the same type belonging to the **pool** ***p1***. The two names are, of course, local to their individual **templates** and arbitrarily chosen. Their types, however, must refer to the same **access interface** if the equivalence is to be valid. Similarly each of the three **ports** of ***subsys_4*** echoes a **port** of the same type belonging to one of its **component activities**.

Remembering that the program coding for our imaginary design has not yet been written, we will now examine some of the **modules** from the **Mascot database** which represent the **templates** and **specifications** we have been discussing in their graphical form. We will begin with ***a_temp_2***, the **template** from which ***a2*** is to be created.

```
ACTIVITY a_temp_2 ;
   REQUIRES sp : send ;
            otp : out ;
            rp : rec ;
END .
```

The local declarations and the program coding which implements this thread of execution will eventually be added after the three **port** specifications. The external interactions of **objects** created from the **template** are limited to those specified in the **access interfaces** ***send***, ***out*** and ***rec***, to which it refers. The corresponding **specifications** take the form illustrated earlier (for***send*** and ***fetch***) and so need not be included here.

The **channel template** ***chan_1***, which depends on another **access interface**, ***fetch***, is represented textually as follows:

```
CHANNEL chan_1 ;
   PROVIDES sw : send ;
            fw : fetch ;
END .
```

When the contents of the **specifications** ***send*** and ***fetch*** have been completed, **access procedures** and private data storage can be added to ***chan_1*** and correspondence established between the procedures and the various interactions provided at the **windows**.

The existence in the **database** of **access interfaces** ***put***, ***trans*** and ***get***, together with any necessary supporting **definitions**, permits the external dependencies of the remaining **template modules** to be validated in the following form:

```
ACTIVITY a_temp_1 ;
   REQUIRES fp : fetch ;
            tp : trans ;
            pp : put ;
END .

POOL pool_1 ;
   PROVIDES pw : put ;
            gw : get ;
END .
```

We are now in a position to inspect the **template** text of ***subsys_4*** itself. It is presented below in its entirety.

```
SUBSYSTEM subsys_4 ;

   PROVIDES gw4 : get ;
   REQUIRES rp4 : rec ;
            otp4 : out ;
            tp4 : trans ;

   USES pool_1, chan_1, a_temp_1, a_temp_2 ;
      POOL p1 : pool_1 ;
      CHANNEL ch : chan_1 ;
      ACTIVITY a1 : a_temp_1 ( fp = ch.fw,
                               tp = tp4,
                               pp = p1.pw ) ;
      ACTIVITY a2 : a_temp_2 ( sp = ch.sw,
                               otp = otp4,
                               rp = rp4 ) ;

      gw4 = p1.gw
END .
```

After the **module** heading, which establishes the **template**'s name, comes what is known as the **specification part**. This defines the dependency of this **module** on the existence of a number of **access interfaces**. In other words it specifies the **subsystem**'s **ports** and **window**. The features of this part should, by now, be entirely familiar.

Then follows what is known as the **implementation part**. This is something new because none of the **modules** we have examined earlier contain any implementation. It starts with a **USES** section which is simply a list of all the **templates** needed to create the **components** of this **subsystem**. There is one for each **activity** and one for each of the **IDAs** and we have already examined all four of them. After the **USES** section the following four lines of the **module** specify what **components** are to be included and how they are to be connected together.

There are to be two **IDA**s called ***p1*** and ***ch*** created from **templates** ***pool_1*** and ***chan_1***, respectively. Examination of these two **templates** shows that the resultant **components** each possess two **windows** which may be referred to as ***p1.gw***, ***p1.pw***, ***ch.sw*** and ***ch.fw***. The connectivity of the **network** is established by using these **window** references in the specifications of the **activity components** which follow on the next two lines of the **module**. These indicate that there are to be two **activities** called ***a1*** and ***a2*** created from **templates** ***a_temp_1*** and ***a_temp_2***, respectively.

The lists in parentheses define the network connections by means of the 'formal = actual' convention. The **port** names on the left of the equivalences (***fp***, ***tp*** and ***pp*** in the case of ***a_temp_1***) are analogous, in this context, to the formal parameters of a procedure. The corresponding 'actual parameters' specify the points in the **network** to which each **port** is to be connected. Where the connection is direct to an internal **window**, a reference to that **window** is given in the form indicated above. Where there is a '**port** to **port**' connection passing out of the **template**, the name of the appropriate **port** on the boundary of the **subsystem** is given. Thus, in the specification of **activity** ***a1***, the connections are:

port ***fp*** <----------> **window** ***ch.fw***

port ***tp*** of ***a1*** <----------> **port** ***tp4*** of **template**

port ***pp*** <----------> **window** ***p1.pw***

and in the specification of **activity** ***a2*** :

port ***sp*** <----------> **window** ***ch.sw***

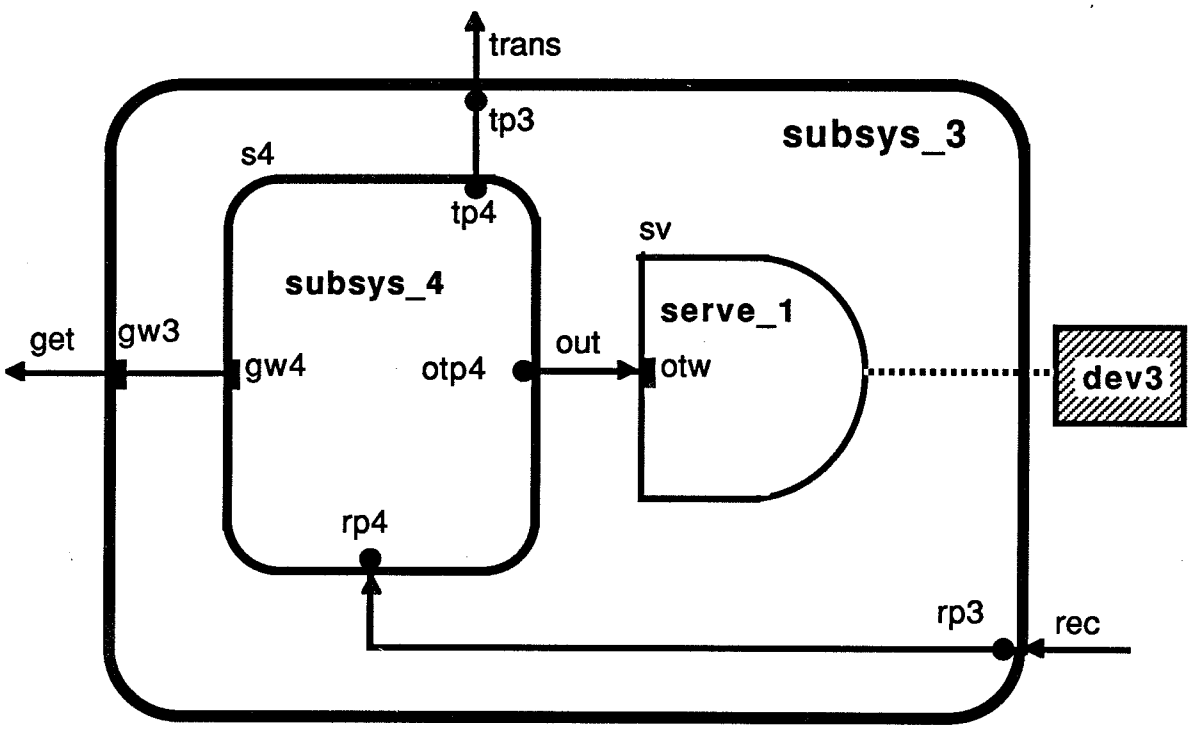
port ***otp*** of ***a2*** <----------> **port** ***otp4*** of **template**

port ***rp*** of ***a2*** <----------> **port** ***rp4*** of **template**

This caters for all the network connections apart from the single ' **window** to **window**' case. The last line of the **module** takes care of this by equating the **subsystem window** with that named ***gw*** belonging to the **IDA** ***p1***. Thus '**port** to **port**' and ' **window** to **window**' connections are dealt with differently in **subsystem templates**. The former are handled through the **activity** 'parameter list' and the latter through equivalence statements.

2.1.6 A Subsystem Containing Another Subsystem and a Server

We are now in a position to consider the **template** text of ***subsys_3*** which utilises ***subsys_4*** to create one of its **components**. Here, again, is the diagram:



The **template** for the second **component** of this **subsystem** in its partially completed state is:

```
SERVER serve_1 ;
   PROVIDES otw : out ;
END .
```

Servers are closely related to **IDAs**. The main difference is that they provide means of interaction with peripherals. Consequently, as well as **access procedures**, which can be invoked by **activities** connected to an appropriate **window**, **servers** may also include **handlers**. These are sections of code which can be connected to hardware interrupts and which are entered for execution, on a pre-emptive basis, whenever the appropriate interrupt occurs. The function of the **handler** is typically to control data transfer and the operation of a hardware device. Transfer between the buffer and active Mascot **components** such as **activities** and **IDAs** is achieved in the normal way via a **path** connected to a **window** of the **server**. This particular **server template** specifies a single **window**, ***otw***, which is of type ***out***.

We have now looked at all the **templates** needed for the **components** of ***subsys_3*** . Here is the **template** of this **subsystem**:

```
SUBSYSTEM subsys_3 ;

   PROVIDES gw3 : get ;
   REQUIRES rp3 : rec ;
            tp3 : trans ;

   USES subsys_4, serve_1 ;
      SERVER sv : serve_1 ;
      SUBSYSTEM s4 : subsys_4 ( rp4 = rp3,
                                otp4 = sv.otw,
                                tp4 = tp3 );

      gw3 = s4.gw4
END .
```

This should readily be understandable in the light of our earlier discussion of **template *subsys_4***. The **specification part** specifies a **window** and two **ports** in terms of **access interfaces *get*, *rec*** and ***trans*** all of which we have already considered. The **implementation part** lists the required **component templates** before specifying the two **components** and their interconnections. There is to be a **server** called ***sv*** created from **template *serve_1*** and a **subsystem** called ***s4*** created from **template *subsys_4***. The connections to the **ports** of ***s4*** are:

port *rp4* of ***s4*** <----------> **port *rp3*** of **template**

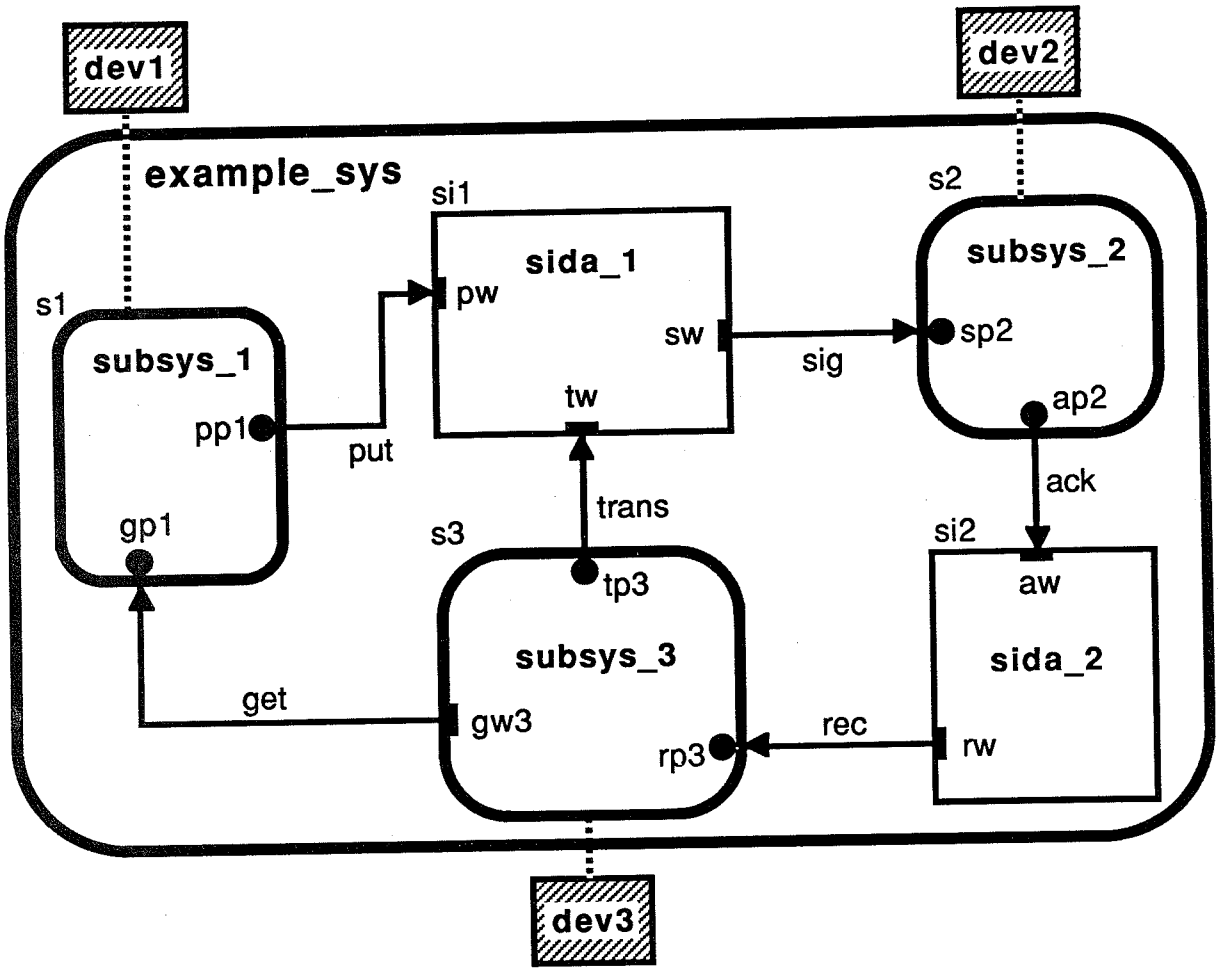
port *otp4* <----------> **window *sv.otw***

port *tp4* of ***s4*** <----------> **port *tp3*** of **template**

The equivalence statement connects the **window *gw4*** of the **component subsystem *s4*** to the **window *gw3*** of the **template**.

2.1.7 A System

Finally in our tour of this imaginary design, we return to our starting point: the **system** diagram.



A **module** to represent a **system template** is very similar to one for a **subsystem**. There are, however, no external network dependencies. In other words a **system** does not specify any **ports** or **windows**. On the basis of what we have already learned, and assuming the existence of the necessary additional supporting **templates** and **specifications**, it is easy to understand **system** ***example_sys*** :

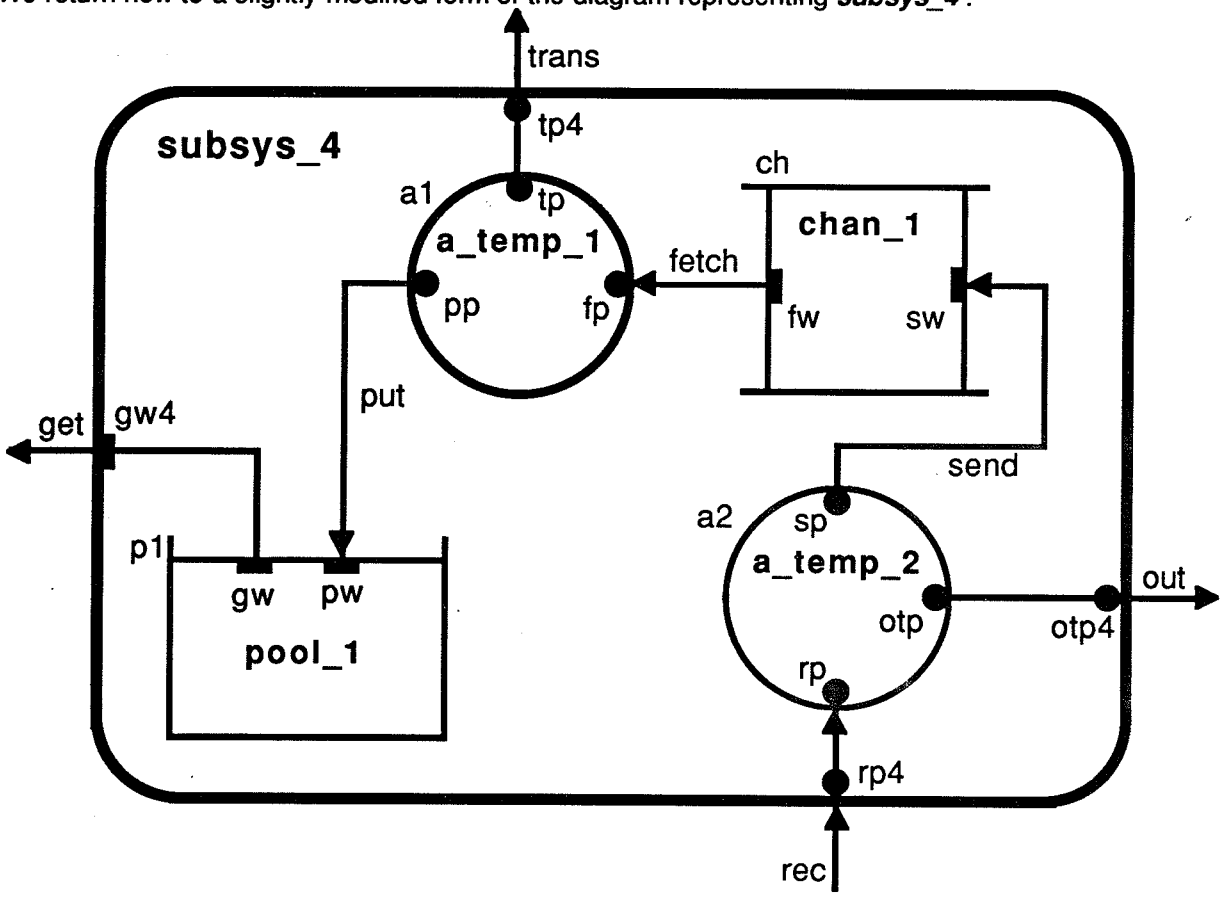
```
SYSTEM example_sys ;

   USES subsys_1, subsys_2, subsys_3, sida_1, sida_2 ;
      IDA si1 : sida_1 ;
      IDA si2 : sida_2 ;
      SUBSYSTEM s2 : subsys_2 ( sp2 = si1.sw,
                                ap2 = si2.aw ) ;
      SUBSYSTEM s3 : subsys_3 ( tp3 = si1.tw,
                                rp3 = si2.rw ) ;
      SUBSYSTEM s1 : subsys_1 ( pp1 = si1.pw,
                                gp1 = s3.gw3 ) ;
END .
```

In this informal survey of the features of the Mascot scheme of design representation we have discussed the function of **ports, windows** and **paths** and examined examples of two kinds of **specification** (**access interfaces** and **definitions**) and five kinds of **template** (**activities, IDAs, servers, subsystems** and **systems**). This selection corresponds roughly to the set of features in the mandatory subset of the definition. Study of the full definition will reveal many extensions to these basic concepts and, in particular, the optional availability of **composite** forms of most of the **templates** and **specifications** presented here in their **simple** form.

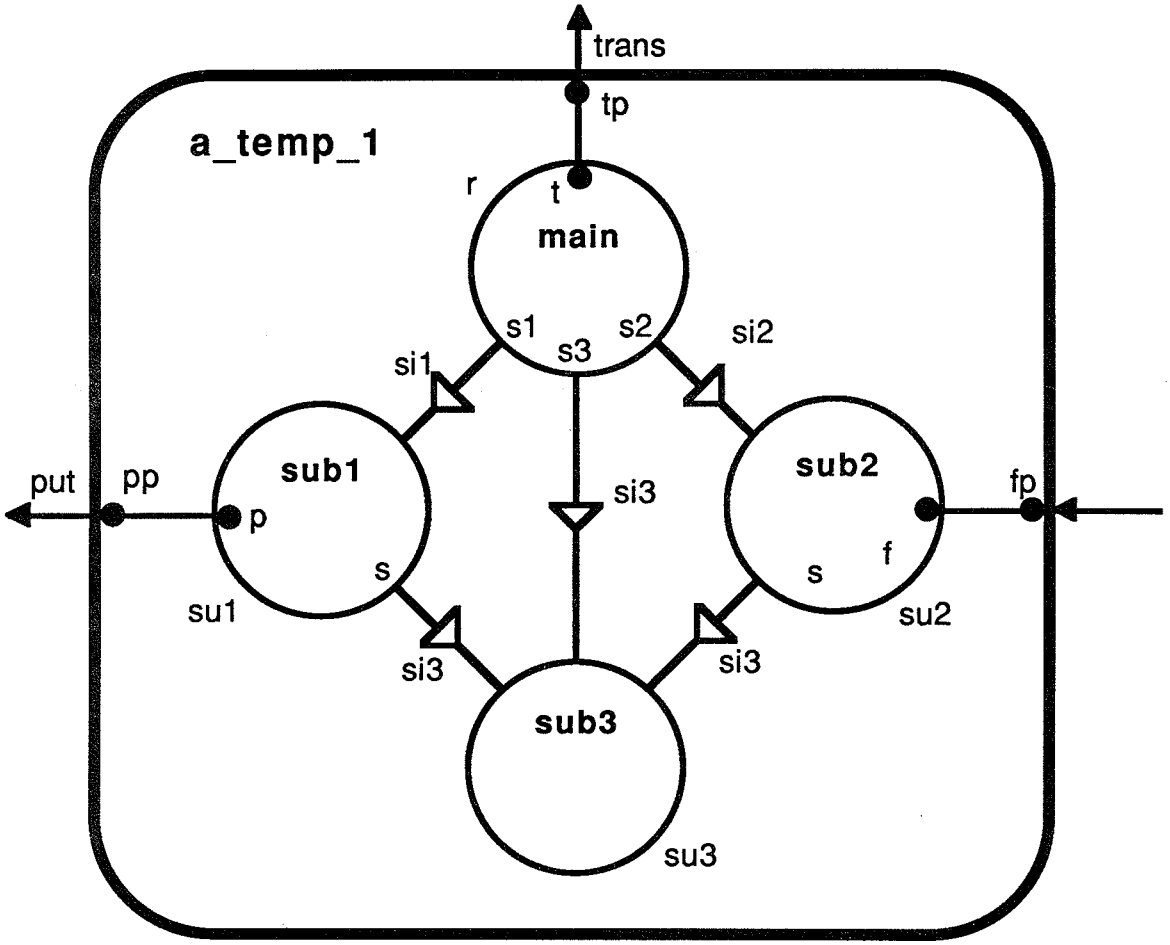
2.1.8 A Composite Activity

A **composite module** is one which is further decomposed in terms of lower level **modules. Systems** and **subsystems** clearly come into this category. They do not, in themselves, introduce coding but merely describe a set of related **components. Composite** forms of **activities, IDAs** and **servers** are described in the following sections. To complete this introduction we will look at just one of these forms: the **composite activity.** This is chosen because it exemplifies sequential decomposition rather than the network decomposition which we have already seen. An **activity** whose implementation is complex may be designed as a set of **components** which, in the executing system, communicate via procedural interfaces. The Mascot design representation provides both graphical and textual conventions to describe this form of construction.

We return now to a slightly modified form of the diagram representing ***subsys_4*** .



This is the lowest level of the hierarchy that we have so far visited. The modification is that the **component** symbol ***a1*** has been drawn with a thick boundary like that for the **subsystem**. The implication is that the corresponding **module** is a **composite** one. To investigate its structure we must expand the symbol to reveal a lower level **template** diagram.



The coding of **component** ***a1*** is to be divided into four separately developed modules. One of them, labelled ***r***, is a design element known as a **root**. Every **composite activity** contains exactly one **root** as this is the **component** which contains the initial entry point of the coding. This particular **root** is created from a **template** called ***main*** which in turn possesses a **port** of type ***trans*** connected, via a **port** to **port** connection, out through the boundary of the **activity template**.

The other three **components**, ***su1***, ***su2*** and ***su3*** are **subroots** created from **templates** ***sub1***, ***sub2*** and ***sub3***, respectively. These are essentially collections of procedures and a **composite activity** may possess any number of them. Their relationships with the **root** and with each other are represented graphically by lines bearing hollow arrow heads and known as **links**. They indicate here that ***r*** calls procedures, directly, in each of the **subroots** and that ***su1*** and ***su2*** both make calls on procedures in ***su3***. Just as the types of **paths** in a **network** take the textual form of **access interfaces**, so **link** types are represented in the **Mascot database** by **specifications** called

subroot interfaces. Here is one of the three needed for the new version of **template *a_temp_1*** :

```
SUBROOT INTERFACE si1 ;
.
.
END.
```

The contents of such **modules**, when complete, will be similar to those of **access interfaces**. Principally, they are used to specify the headings of procedures which **subroots** make available to the **root** and to other **subroots**.

We can now examine the **template**, ***main***, from which the **component *r*** is created.

```
ROOT main ;
   REQUIRES t : trans ;
   NEEDS    s1 : si1 ;
            s2 : si2 ;
            s3 : si3 ;
END .
```

The **port** is specified exactly as in a **simple activity**. The **NEEDS** section specifies the **links** which are connected to **components** of this type in terms of the **subroot interfaces**. Coding will, of course, be added to this **module** before the final **END**.

We now turn to the **template**, ***sub1***, for **component *su1***. This is a **subroot** possessing a **port** together with **links** both entering and leaving.

```
SUBROOT sub1 ;
   REQUIRES p : put ;
   GIVES si1 ;
   NEEDS s : si3 ;
END.
```

The new feature here is the **GIVES** section. It specifies the **subroot interface** which this **subroot** implements. Every **subroot** implements precisely one such **interface**.

Template *sub3* which, however, possesses no **ports** and no outward going **links**. It provides procedural interactions but does not make use of any.

We have now examined all the **specifications** and **templates** that are needed for the design of the **composite activity *a1***. Here, in its complete form, is the **template *a_temp_1***.

```
ACTIVITY a_temp_1 ;
   REQUIRES fp : fetch ;
            tp : trans ;
            pp : put ;

   USES main, sub1, sub2, sub3 ;
      SUBROOT su3 : sub3 ;
      SUBROOT su1 : sub1 ( p = pp,
                           s = su3 ) ;
      SUBROOT su2 : sub2 ( f = fp,
                           s = su3 ) ;
      ROOT r : main ( t = tp,
                      s1 = su1,
                      s2 = su2,
                      s3 = su3 ) ;
END.
```

Following the **specification part**, identical with that of the earlier version, comes the **implementation part**. This begins with a **USES** section which lists the **root** and **subroot templates** needed to create the **components**. Then follows a specification of each **component** with its network and subroot connections. The former are all **port** to **port** connections and are dealt with in the same manner as in **subsystems**. The remaining parameters specify which **subroots** implement each of the the **subroot interfaces** specified in the corresponding **NEEDS** sections.

With this discussion of the Mascot facilities for the sequential decomposition of individual threads of execution we come to the end of this informal introduction to the Mascot scheme of design representation. All the principal concepts have been discussed and, if they have been understood, the formal definition which follows should present no insuperable difficulties.

In the remainder of this chapter all the design representation features of Mascot, mandatory and non-mandatory, are discussed. Together with the relevant parts of Appendices A, D and E, it constitutes a formal definition of this aspect of the method.

2.2 A NOTE ON PRESENTATION STRATEGY

In determining the order in which topics should be presented in this part of the Handbook it was considered important to focus particular attention on the subset of features mandatory in any Mascot 3 implementation. Not only are these features presented first but they are described without reference to the existence of the remaining features which make up the full definition. As a consequence of this strategy many of the diagrams defining the syntax of the Mascot design representation language appear, in the first few sections, in a simplified form. Where this is the case the simplified diagram has been identified as such by means of a rectangular frame. There are features of these framed diagrams whose significance may not be immediately clear. These include the apparently gratuitous use, for example, of the word ***simple*** in the name of a syntactic entity or the employment of an entity which expands directly into another. These matters can be clarified by reference to Appendix A. The mandatory subset is identified from the full set of Mascot features in Appendix E.

2.3 PATHS, PORTS AND WINDOWS

Description

The Mascot method addresses the problems of creating application software which consists of collections of communicating, parallel processes executed on processor configurations of arbitrary complexity. For any particular Mascot application, the software design is embodied in a set of hierarchically related **templates**. At the lowest level of each branch of the hierarchy, **element templates** provide patterns for fundamental software **objects** capable of performing either a data processing or a data communication function. A **path** between a pair of these entities denotes the existence of a set of operations (usually involving data transfer) provided by one of the participants for use by the other. All such transactions thus involve a passive component, concerned with information storage and transmission, and an active component, concerned with information processing. The nature of the interactions associated with any particular **path** is defined, in procedural terms, by a textual unit (**module**) called an **access interface.** Thus an **access interface** is a **specification** defining a set of operations, implemented by the passive component and which, when invoked by an active partner, transmit data in either or both directions.

Data-type definitions and constants relevant to these interactions may also be supplied to both participants from the **access interface**. By associating definitions with a communication route in this way, conformity with the semantic rules of strongly typed implementation languages is made possible and consistency of usage between separately created communicating **objects** is ensured. This data-typing information is held separately in another kind of **specification**, known as a **definition**, from where it can be incorporated into those **access interfaces** which require it.

Every **path** in a Mascot network is connected at one end to a **port** of a **component. Ports** are normally possessed by **activities**, which are the fundamental processing **elements** of a Mascot **system**, and by **subsystems** which may fulfil a similar role at a higher level of the design. They may, however, also occur in the fundamental communication **element**, the **IDA**. A **port**, as specified in a **template**, is a named reference to an **access interface**. It expresses, in terms of the **access interface**, a requirement to invoke data transfer (and other) operations which are implemented outside the **component** in which the **port** is specified. Establishing, in the specification, the type of the **port** to which a **path** is to be connected, thus determines the group of operations which are required.

At the other end of each **path**, in a communications component (usually an **IDA** or a **subsystem** performing a data communications role) is a **window**. A **window** as specified in a **template** is, like a **port**, a named reference to an **access interface**. It specifies, in terms of the **access interface**, a set of operations to be provided. These operations may be invoked by other components connected to the

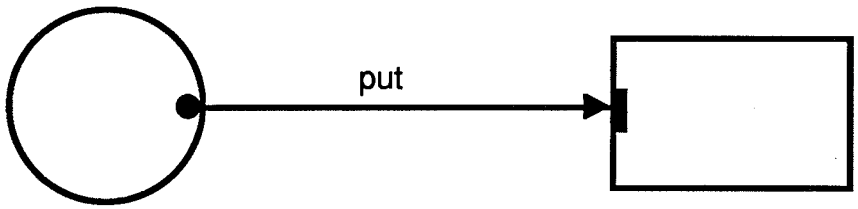
window. Establishing, in the specification, the type of the **window** to which a **path** can be connected, thus determines the group of operations which are provided. Notice that whereas each **port** in a **network** is connected by a single **path** to the provider of the interactions which are required, any number of **paths** may be connected to a **window**. This is to say that any number of **components**, which possess matching**ports**, may invoke the interactions provided at a particular **window**.

Both **activities** and **IDAs** may be used at any depth within an hierarchy of **subsystems**. In the **network** diagram, a **port** may therefore be separated by the boundaries of a number of enclosing symbols, from the source of the operations for which it expresses a requirement. A **window** may similarly be separated from the nearest **port** along a given **path**. In these circumstances, the requirement expressed by the **port** and the provision expressed by the **window** are propagated along the communicating **path** through a series of identical **ports** and **windows** established in the higher level constructs .

Graphical Representation

The interactions between components readily lend themselves to diagrammatic representation in terms of static data flow **networks** in which the node symbols represent either individual **elements** or lower level **networks**. A **path** is the route along which information is transmitted from one particular **element** or **network** to another. Its diagrammatic representation is a thin line connecting the two symbols together. This is normally labelled with the name of the **access interface** which constitutes the type of the **path** and carries a solid arrow head to indicate the direction of data flow.

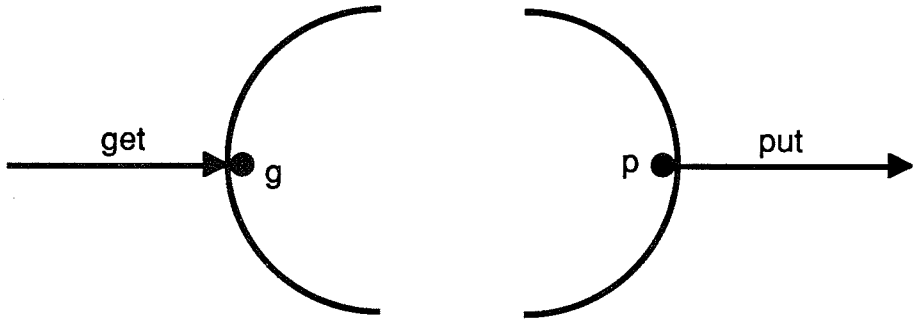
The diagram below shows an active **component**, on the left, connected via a **path** to a passive **component**, on the right, with a **port** at the active end of the **path** and a **window** at the passive end.



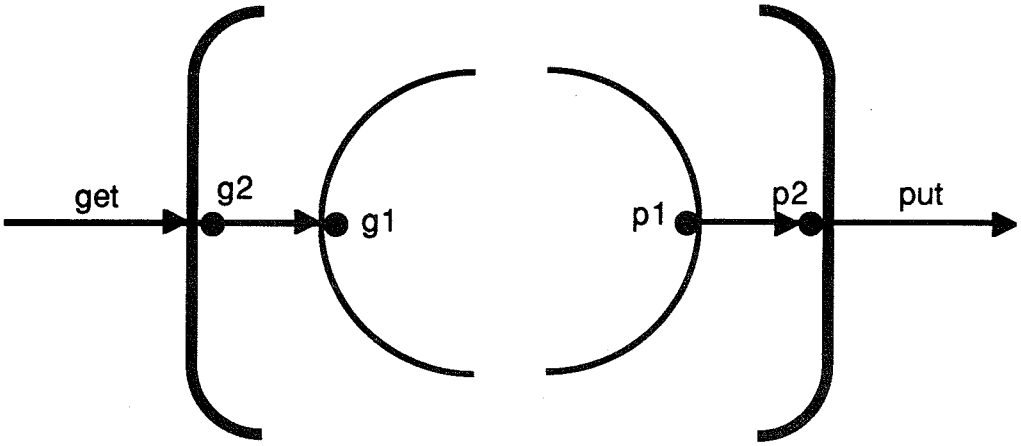
Thus a **port** is represented in a Mascot **network** diagram by a small filled circle connected to a line representing a **path**. The **port** symbol may be labelled with an identifier, unique within the **template**

which possesses it, which is used to distinguish between **ports**. The **path** itself, as we have seen, is labelled with the name of the **access interface** which defines its type so that, from an external point of view, a **port** represents a network connectivity constraint. As **ports** may function as either sources or sinks of data, the solid arrow head which indicates the direction of data flow may point either away from a **port** or towards it.

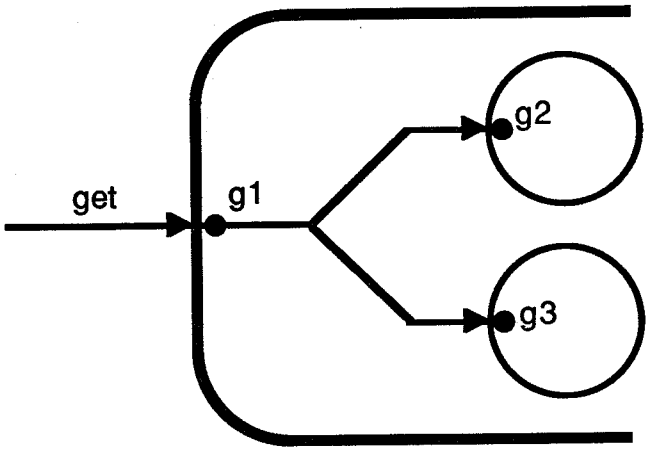
On a Mascot **network** diagram, a **port** symbol is placed just inside the boundary of the **template** or **component** to which it belongs.



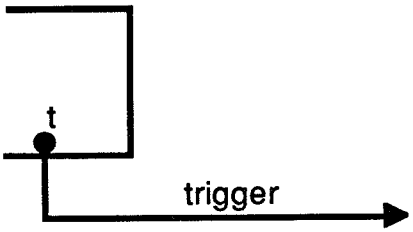
In cases where data are transmitted, along a **path**, across one or more boundaries, the **port** symbol is repeated at each boundary. This reflects the necessity of respecifying the **port** in the text of each **template**.



Different **activities** at the same or different levels of the hierarchy may share a set of operations provided by a common source. The diagram may be simplified in such cases by merging the **paths**. This is illustrated in the diagram below which shows two **activities** reading data from a common source and incorporated, at the same level, within a **subsystem**.

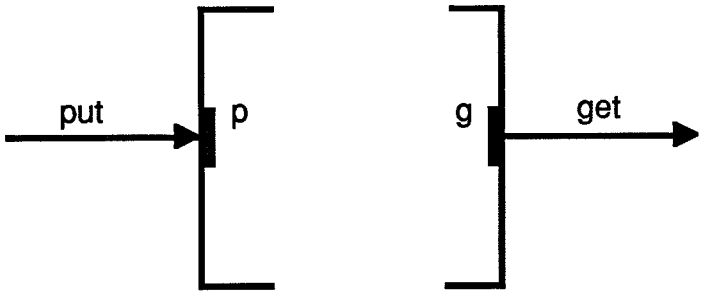


Performing the operations, defined by means of a **port** and its associated **access interface**, demands the execution of code. This code is contained in the passive communication **element** of a Mascot **system**, the **IDA**, and is executed in response to direct invocation by one of the active processing **elements**. On occasions, especially where distributed processing hardware is involved, one passive component may need to use the operations implemented by another in order, for example, to propagate information from one hardware unit to another. It is thus necessary that communication components should be permitted to possess **ports** so that this type of requirement may be met. Despite the presence of a **port** on the boundary of the symbol shown below, it nevertheless represents a passive component and consequently is drawn with square corners.

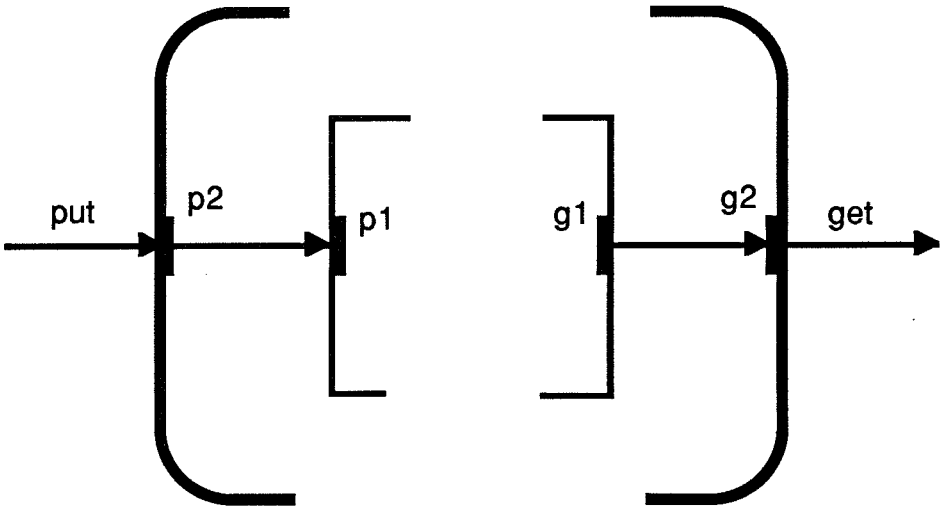


A **window** is represented in a Mascot **network** diagram by a thin, filled rectangle connected to a line representing a **path.** The **window** symbol may be labelled with an identifier, unique within the **template** which possesses it, which is used to distinguish between **windows**. The **path** itself, as we have seen, is labelled with the name of the **access interface** which defines its type so that, from an external point of view, a **window**, like a **port**, represents a network connectivity constraint. As **windows** may function as either sources or sinks of data, the solid arrow head which indicates the direction of data flow may point either away from a **window** or towards it.

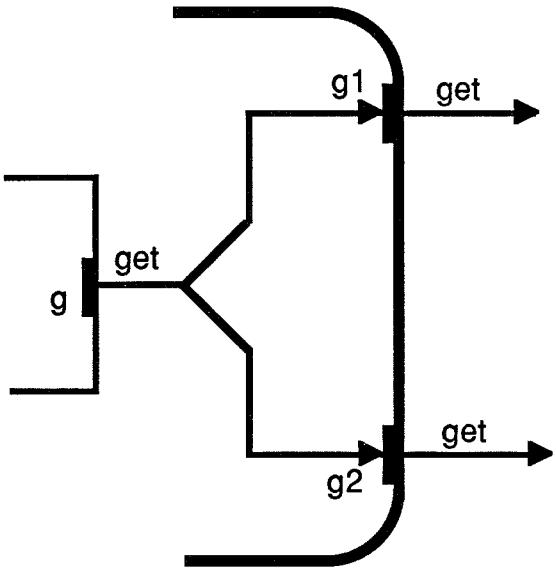
On a Mascot **network** diagram, a **window** symbol is placed just inside the boundary of the **template** or **component** to which it belongs.



In cases where the operations provided transmit data along the **path**, across one or more boundaries, the **window** symbol is repeated at each boundary. This reflects the necessity of respecifying the **window** in the text of each **template**.

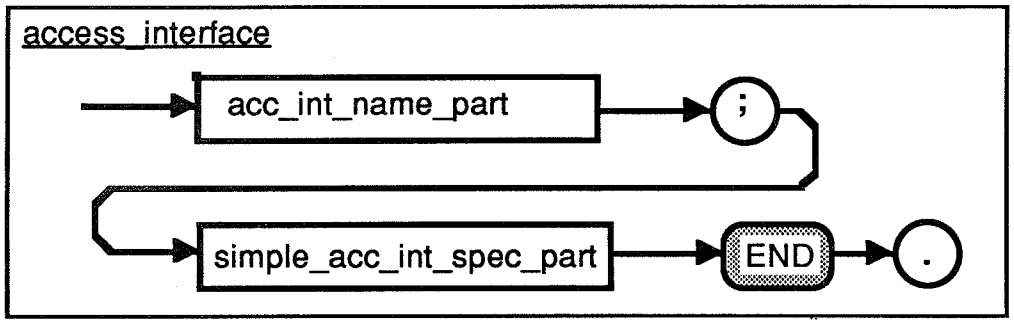


An **IDA** inside a **subsystem** may provide operations to satisfy the common requirements of several **port** bearing **activities** or **subsystems**. The diagram may be simplified in such cases by merging the **paths** together. This is illustrated in the diagram below.

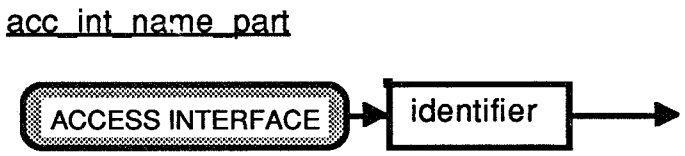


Textual Representation

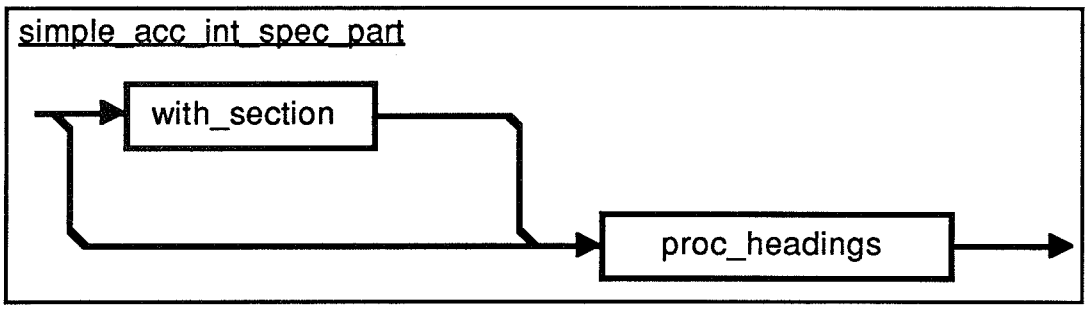
The **module** which defines the operations, required by a **port**, provided by a **window**, and hence defines the interactions along a **path** is the **access interface**. This is a **specification**. In the mandatory subset of the Mascot definition, the contents of an **access interface** are procedure headings. These provide sufficient information for interactions to be invoked by an active **component**. For each procedure heading in the **interface**, a corresponding complete procedure, known as an **access procedure**, appears in the **template** for the passive **component**. The syntactic structure is shown in outline below:



It consists of a ***name part*** , which establishes the **class** of the **module** and gives the **specification** a name:



and a ***specification part*** which takes the following form:



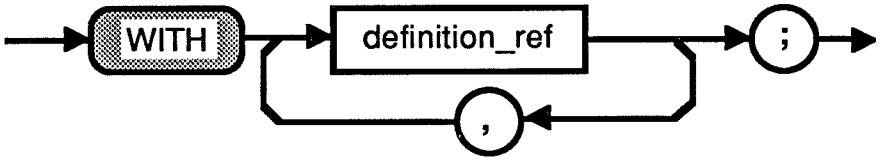
The ***specification part*** optionally begins with a ' **WITH** section' which expresses the dependency of this **module** on other **modules**. The simplest **interfaces** (those which refer only to data-types which are basic to the implementation language) contain no dependencies, and the ***specification part*** then reduces to a set of procedure headings.

The following is an example of a **simple access interface** with no external dependencies.

```
ACCESS INTERFACE send_1 ;
   PROCEDURE secure ;
   PROCEDURE release ;
   PROCEDURE transmit ( i : integer ) ;
END .
```

The ' **WITH** section' of an **access interface** contains a list of references to **definition specifications** and so implements the mechanism, referred to above, by which definitions may be shared by several **interfaces**.

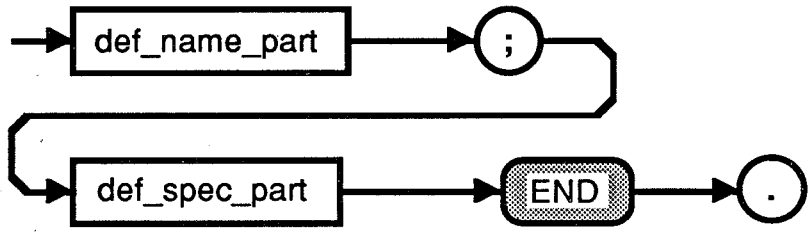
with_section



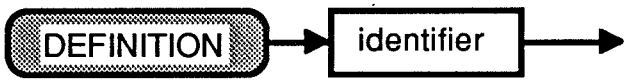

```
WITH flow_data, table ;
```

The syntactic structure of a **definition**, which has no graphical representation, is as follows:

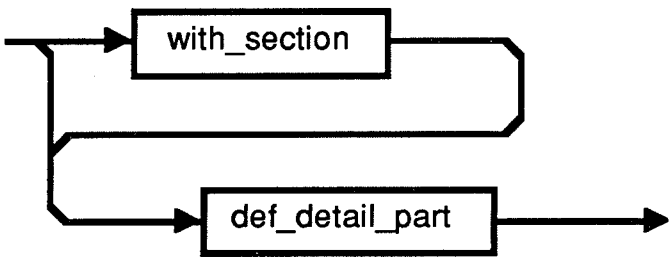
definition



def_name_part



def_spec_part



The content of a **definition module** is limited to the definitions of symbolic constants and data-types. It is beyond the scope of this Handbook to define precisely how types are defined in a **definition module** and how these types can be used by other **modules**, since this impinges on the syntax and semantics of the implementation language selected. However, a particular implementation may:

(1) insist that any data-type used in the **Interface** will be defined in a **definition module** of the same name and named in the ' **WITH** section', or

(2) allow any data-type used in the **Interface** to be undefined, on the assumption that it will be defined in one of the **definition modules** named in the ' **WITH** section', or

(3) insist that any data-type used in the **Interface** is already defined in one of the **definition modules** named in the ' **WITH** section'.

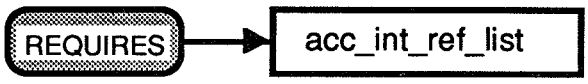
The following examples illustrate the use of **definitions**.

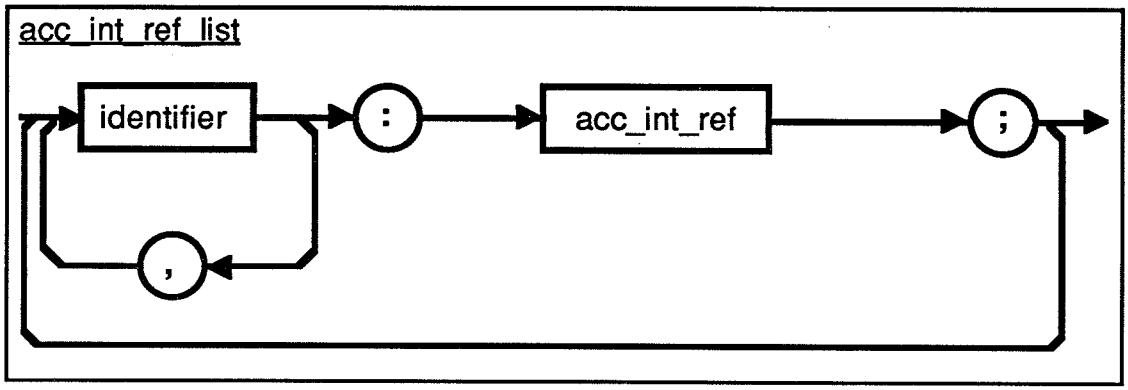
```
DEFINITION colour ;
   TYPE
      colour = (red, green, yellow, blue) ;
END .

DEFINITION palette ;
WITH colour ;
   CONST
      max = 10 ;
   TYPE
      palette = ARRAY[1 .. max] OF colour ;
END .

ACCESS INTERFACE spectrum ;
WITH palette ;
   PROCEDURE paint( shade_card : palette ) ;
END .
```

Ports

The textual representation of a **port** is shown, in syntactic form, below:

port_spec





The following are examples of individual **port** specifications:

```
REQUIRES g : get;
         pa, pb : put;
```

where ***get*** and ***put*** are the names of **access interfaces** defining two sets of operations required in this **template**. The formal **port** names ***g***, ***pa*** and ***pb*** are used for two main purposes. First they are used, in **network modules**, to express the connectivity of the **components**. Second, in a **simple module**, they must be used to identify a particular **port** procedure. The syntax of such a reference is as shown below:

where the procedure identifier after the '.' is specified in the **access interface** which defines the type of the selected **port**.

Windows

The textual representation of a **window** is shown, in syntactic form, below:

window_spec

where the form of the list is the same as that shown above for **port** specifications.

The following are examples of **window** specifications:

```
PROVIDES g : get ;
         pa, pb : put ;
```

where ***get*** and ***put*** are the names of **access interfaces** defining two sets of operations which this **module** makes available to the outside world. The implication of these specifications is that the **module** implements, directly or indirectly, the procedure headings in the bodies of the **access interfaces** ***get*** and ***put***.

The formal **window** names ***g***, ***pa*** and ***pb*** are used for two main purposes. First, they are employed, in **network modules**, to express the connectivity of the **components**. Thus a construction such as:

is used to refer to a specific **window** of a specific **IDA component**.

Second, the following construction:

(similar to that already discussed in connection with **ports**) may be used in establishing the correspondence between **access procedures** and the procedures made available at specific **windows**. References of this type are to be found in the ***access equivalence lists*** described in Section 2.6.

2.4 SYSTEMS AND SUBSYSTEMS

Description

Mascot supports progressive development of a design, from outline to completion, with checks performed for logical consistency at each stage. The detailed arrangements whereby this is achieved are described in Section 3.1. In this section we examine the design feature which makes hierarchical, top-down design expression possible in Mascot: the concept of **systems** and **subsystems**. These design elements perform a constructional role. Known formally as **networks**, they define, for part (**subsystem**) or all (**system**) of a Mascot application, the components which are to be created and how they are to be connected together.

In choosing the order of presentation of topics in this Definition, it has been judged desirable that the decompositional aspects of Mascot should be treated before embarking on detailed descriptions of the various forms of system component. Inevitably, then, this section is concerned with the construction of systems from building blocks whose nature remains to be fully revealed. This should present no insuperable difficulties provided that the general principles covered in Section 2.1 have been understood.

In its initial stages, a Mascot design may consist simply of a set of **subsystems** representing the principal functional or geographical units of the application. Since **subsystems** may possess **ports** and **windows**, they can be interconnected by **paths** which, through their associated **access interfaces**, define the nature of the interactions that are to take place. Subsequently, the details of the **subsystems** are added in the form of **templates** for **networks** of **activities**, **IDAs** and **servers** together with other, lower level, **subsystems**. It is this last feature, the ability to include **subsystems** within **subsystems** to any depth, which facilitates an hierarchical form of design expression. As a design entity a **subsystem** may play the role either of a processing or a communication element or of a mixture of the two.

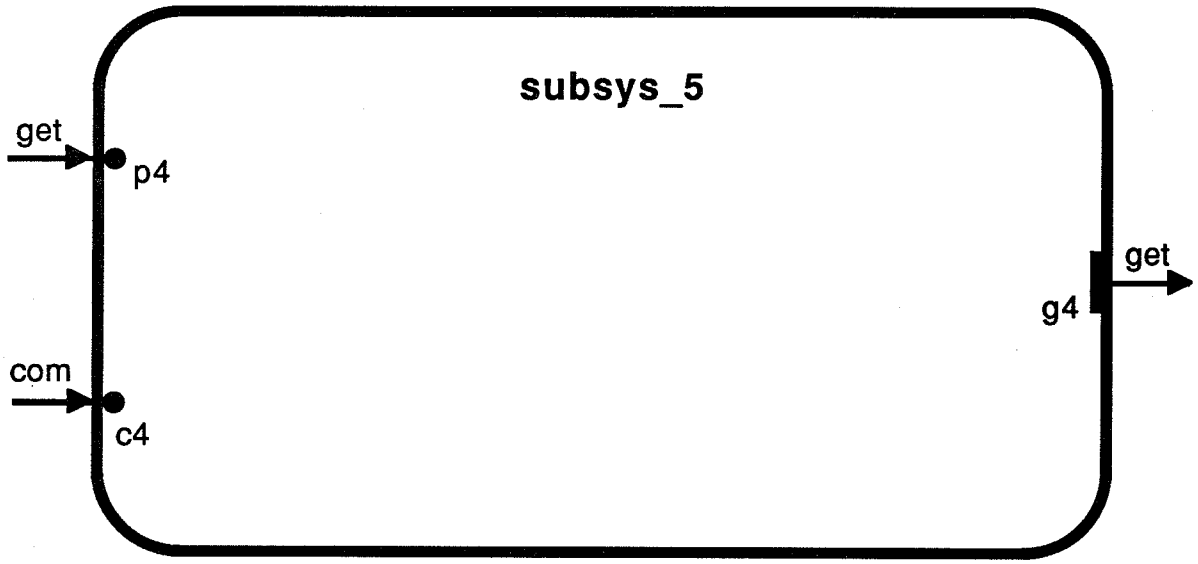
As will be seen below, the graphical representation of a **subsystem** would lead one to suppose, on the basis of a familiarity with the rest of the Mascot graphical conventions, that it necessarily incorporated one or more independently scheduled threads of execution. However exceptions can arise. Since the existence of a **subsystem** is established before its internal design is carried out, there is always the logical possibility that subsequent decomposition will show that no component **activities** or **subsystems** are required. This is an allowable form and, where it arises, the **subsystem** is functionally identical to either a **composite IDA** or a **composite server** (both of which are the subject of later sections).

A **system** is a **network** which encompasses the whole of the application. Explicitly or implicitly, it constitutes a complete description of the software. It is the highest or outermost level of the hierarchical design expression. Consequently it differs from a **subsystem** only in having, by definition, no external dependencies other than those which may be satisfied during system building (see later Section 2.8). All communication with hardware or software objects in its external environment is implicit. In the remainder of this section, attention will be directed largely to **subsystems** of which the **system** will be treated as a special case.

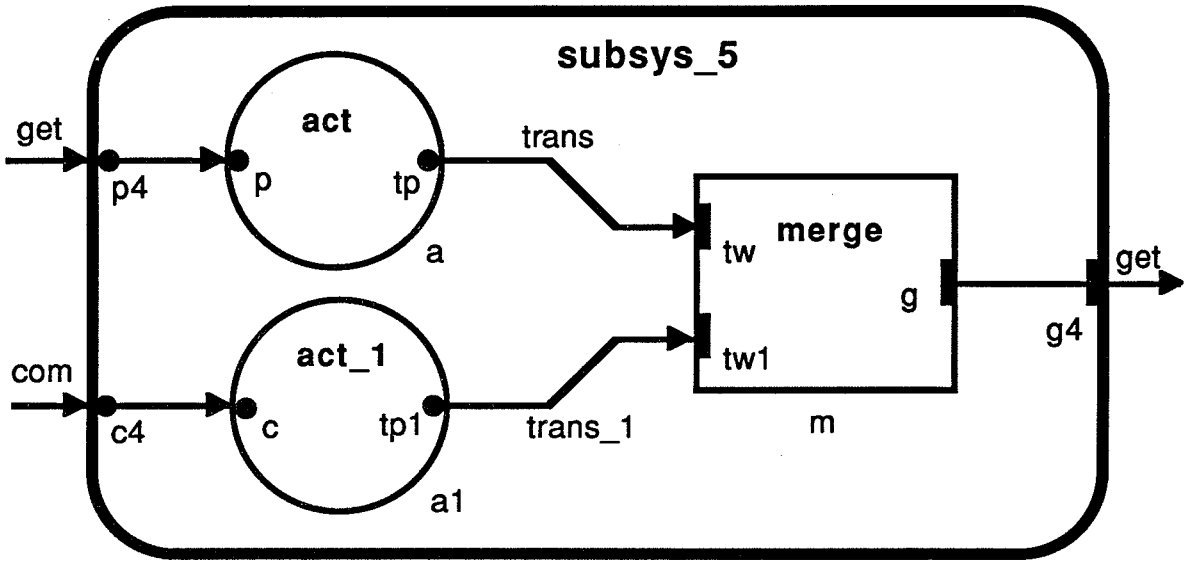
Graphical Representation

A **subsystem** is represented in a Mascot **network** diagram by a closed curve with rounded corners. As it is by nature a **composite** entity, it is drawn as a thick or double line or in such other distinctive form as may have been adopted as a suitable convention. The shape is usually that of rounded rectangle. External dependencies are shown by lines which pass through the boundary of the symbol and where these represent **paths** they terminate in either a **port** or a **window** symbol.

The diagram below shows a **subsystem** as it might appear in the early stages of design. Its external dependencies are determined but not its internal structure.

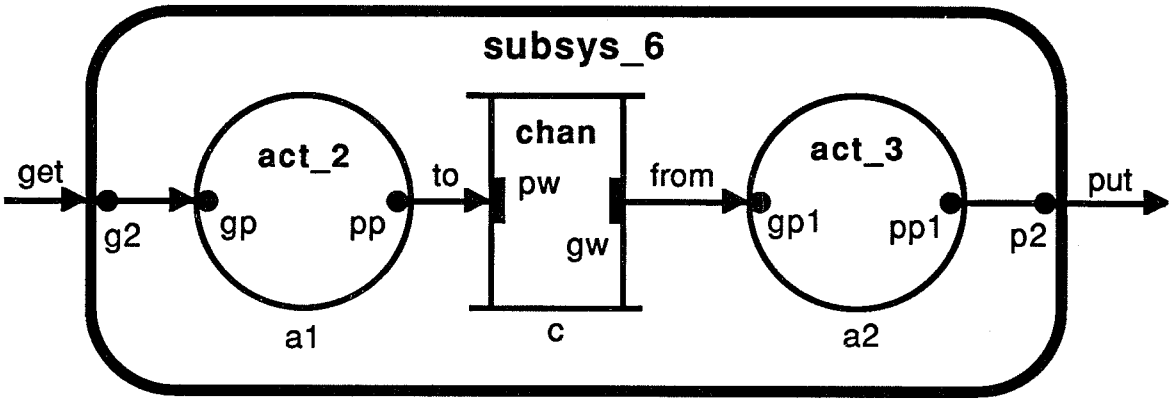


This particular example possesses two **ports** and a **window** and its purpose is to process two input data streams to produce a single output stream. The next diagram displays the internal structure of the **template** ***subsys_5***.

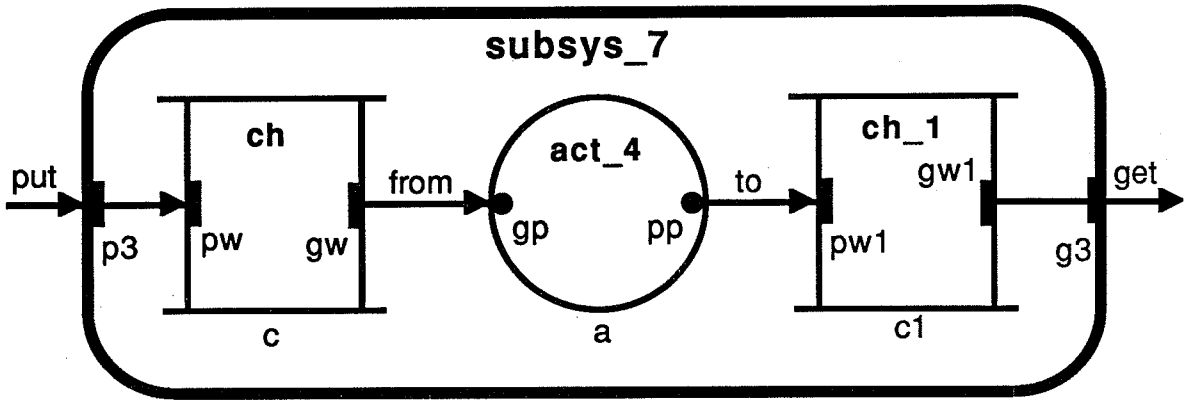


Two **activities**, ***a*** and ***a1***, created from **templates** ***act*** and ***act_1*** respectively, are connected to the two externally visible **ports** which evidently receive the two input streams. Each **activity** is connected via a separate **path** to a separate **window** of the **IDA** ***m***. These two internal **paths** are represented by distinct **access interfaces**, ***trans*** and ***trans_1***. Finally it is **IDA** ***m***, created from the **template** ***merge***, which provides the externally accessible **window**.

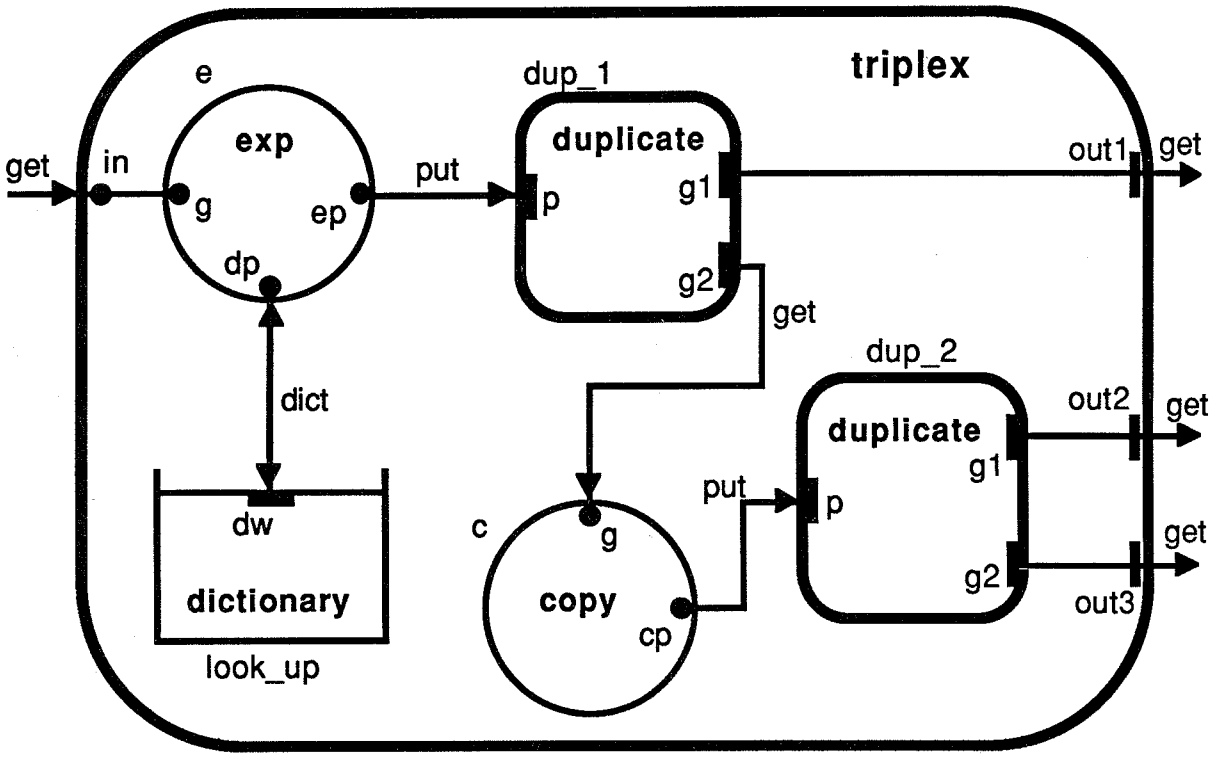
subsys_5 is a simple example but is relatively general in possessing both **ports** and **windows**. ***subsys_6*** below, on the other hand, has only **ports** and therefore externally resembles an **activity**.



Conversely, ***subsys_7*** externally resembles a pure communication element.



As a final graphical example, ***triplex***, to seasoned Mascot users an old friend in a new guise, illustrates a **subsystem** with embedded **subsystems** in its structure.



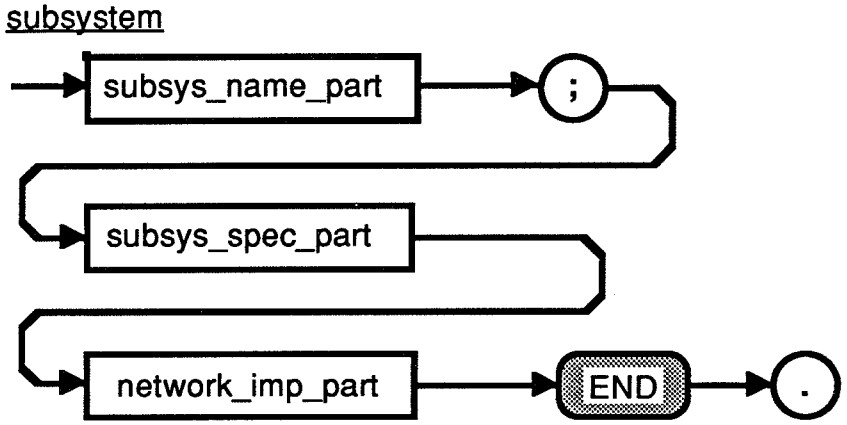
The thick lines of the symbols ***dup_1*** and ***dup_2*** reveal these **components** to be of a **composite** nature. They are **subsystems** and, furthermore, they are both instances of the same **template**, ***duplicate***. Notice that the design would have to be expanded to at least another level down to reveal the **components** which actually provide the facilities, defined in **access interface** ***get***, provided at the three **subsystem windows**. ***triplex*** itself possesses a **port**, ***in***, and three **windows**, ***out1***, ***out2*** and ***out3*** all of type ***get***.

Mascot 2 users may be inclined to wonder at the complication of the well known elementary example of ***triplicated_expansion***. Under what circumstances would it be necessary to use a **subsystem** rather than an eminently simple **activity** to duplicate a data stream without transformation (for such is the traditional version of this example)? The explanation arises from the applicability, mentioned in the

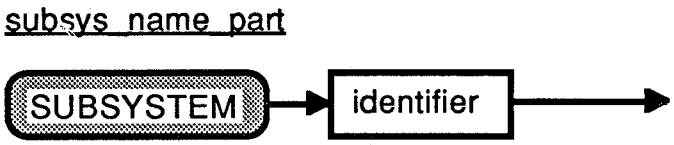
Introduction, of Mascot 3 to very large distributed computing systems. If each of the three data streams associated with the two instances of **subsystem template** ***duplicate*** fall under the jurisdiction of different processors with private and shared memory units, something considerably more complicated than a two-statement **activity** may be needed.

Textual Representation

The syntactic structure of a Mascot **subsystem template** is shown, in outline, below.



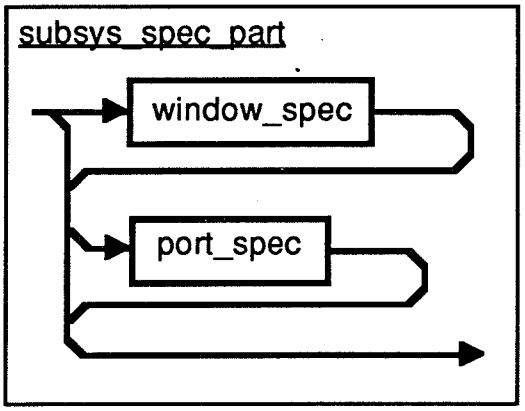
The ***name part*** establishes the **class** of the **module** and gives the **template** a name.



For example:

SUBSYSTEM subsys_5 ;

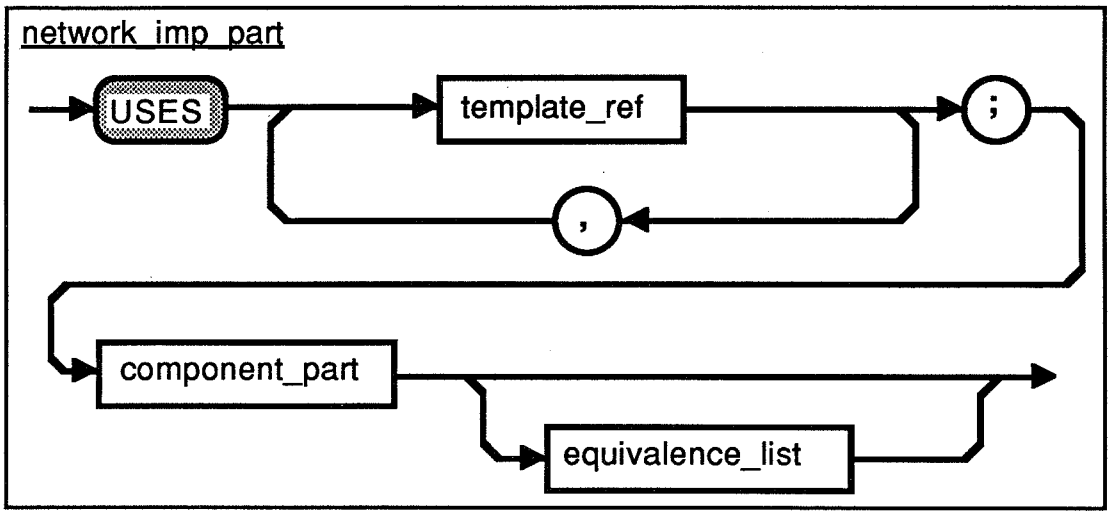
The ***specification part*** specifies the network dependencies. Its syntax, in the mandatory subset, is defined in the diagram below.



The form of **port** and **window** specifications is defined in the preceding section of the Handbook. For **subsystem** ***subsys_5*** the ***specification part*** is:

```
PROVIDES g4 : get ;
REQUIRES p4 : get ;
         c4 : com ;
```

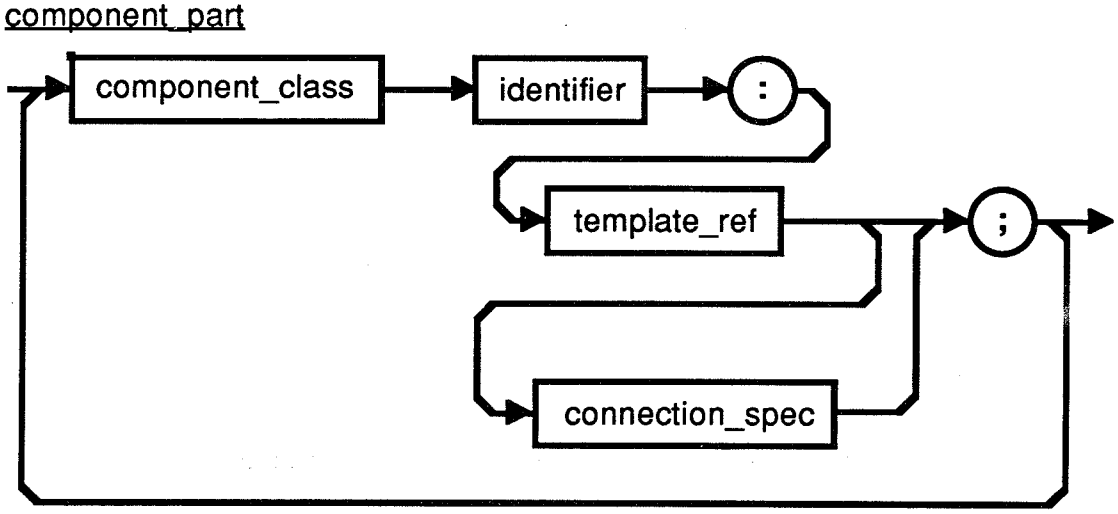
An important point to understand is that the **subsystem template** contains no algorithmic coding. It merely defines the nodes and interconnections of a **network**. As explained above, the **modules** which represent its component parts, the nodes, are **activities, IDAs** and **servers** together with other **subsystems**. The interconnections are **paths** whose types are defined by **access interfaces**. All the information needed to create the **components** and connect them together is provided in the **module's *implementation part***. Its syntactic structure is shown below.



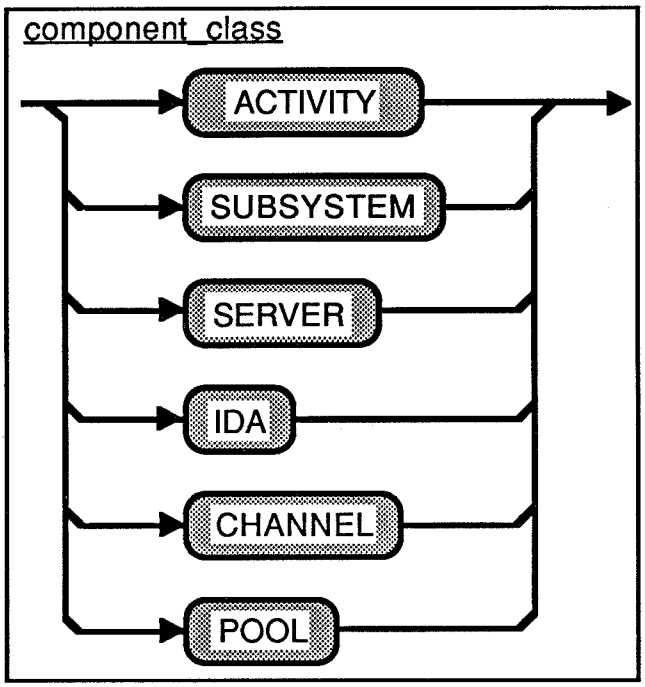
It begins with a **USES** section which lists all the **templates** needed for the **components** of the **subsystem**. This list may not include a **system template**.

```
USES act, act_1, merge ;                    { subsys_5 }
USES act_2, act_3, chan ;                   { subsys_6 }
USES act_4, ch, ch_1 ;                      { subsys_7 }
USES exp, copy, duplicate, dictionary ;     { triplex }
```

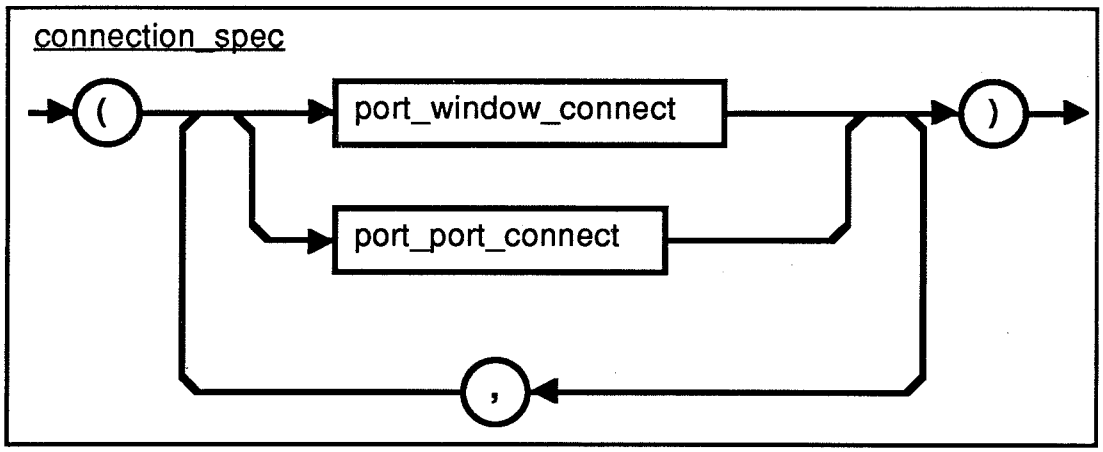
Next comes the ***component part*** in which **components** derived from the **templates** mentioned in the **USES** section are specified.



There are, including the two special varieties of **IDA**, six different **classes** of **component** which can be included in a **subsystem**.

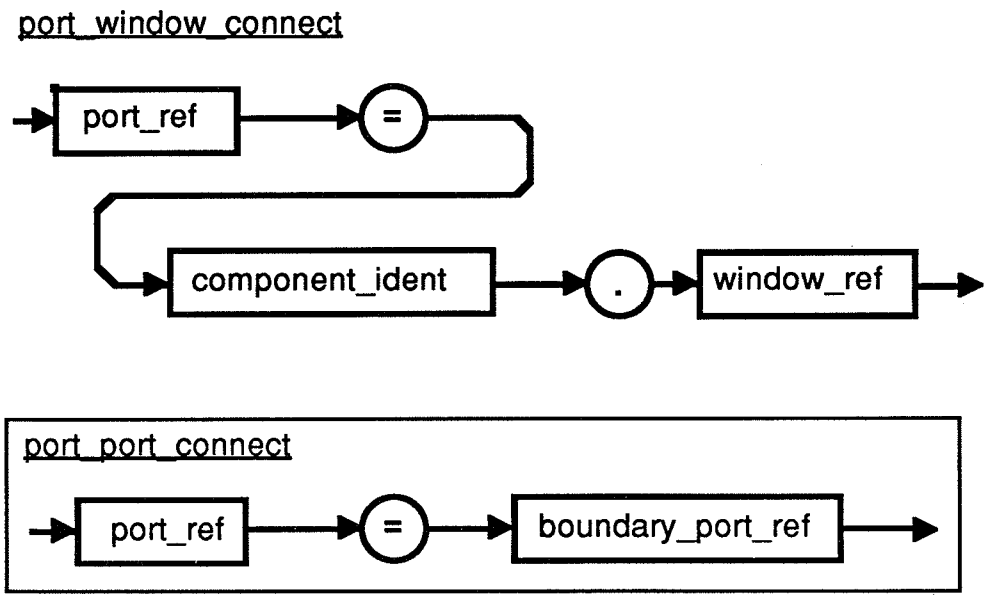


Each **component** is individually introduced by the appropriate language word : **ACTIVITY**, **IDA**, **CHANNEL**, **POOL**, **SERVER** or **SUBSYSTEM** and given a name. These names are the **component** names that, in the graphical form, appear just outside the corresponding symbols. The **template** reference which follows is the name of the **template** from which the **component** is to be created. The **component class** must be the same as the **class** to which this **template** belongs. Where there are **ports**, their connections must be specified in the following manner:



This is equivalent to a set of actual parameters in the call of a procedure. To continue the analogy, the **ports** specified in the **component templates** correspond to the formal parameters. For every **port** a connection to a matching **window** must be established, either explicitly or by way of another matching

port on the boundary ot the enclosing **template**.



Thus the interconnection information consists of a list of identities expressed in the 'formal = actual' convention. Where the connection is an internal one, the reference is to an explicit internal **window**. **Window** names must be qualified with the name of the **component** which provides them. Where the connection is an external one, the 'actual parameter' is the formal name of a **port** on the boundary of the **subsystem**.

Returning to the **subsystem** diagram for ***triplex***, it is now possible to derive the equivalent textual representation. Its ***specification part*** is:

```
PROVIDES out1, out2, out3 : get ;
REQUIRES in : get ;
```

and its **components** are :

```
POOL look_up : dictionary ;
SUBSYSTEM dup_1 : duplicate ;
SUBSYSTEM dup_2 : duplicate ;
ACTIVITY e : exp (    g = in,
                      ep = dup_1.p,
                      dp = look_up.dw);
ACTIVITY c : copy (   g = dup_1.g2,
                      cp = dup_2.p) ;
```

The **port** specifications of the ***exp*** **template** are:

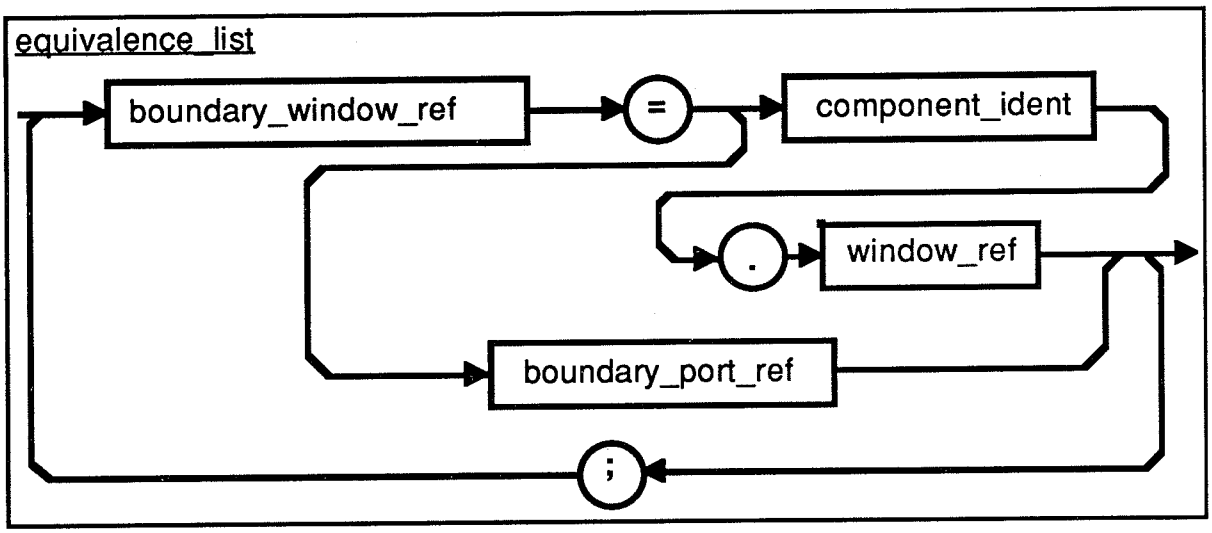
```
REQUIRES g : get;  ep : put ; dp : dict ;
```

and that of the **template** ***copy*** :

REQUIRES g : get ; cp : put ;

Notice that the **port** to **port** connection is represented by specifying ***in*** as the connection corresponding to the **port** ***g*** of **activity** ***e***. That is, instead of specifying an explicit **window** to satisfy this requirement, the **port** of the **activity** is identified with that of the enclosing **subsystem**. A corresponding **window** would have to be provided within any **system** or **subsystem** having a ***triplex*** **component**.

The next aspect of **subsystem templates** to be discussed is the representation of (boundary) **window** to (internal) **window** and (boundary) **window** to (boundary) **port** connections. These are dealt with by means of an ***equivalence*** list (not to be confused with the ***access equivalence list*** used in **IDA templates** to establish a correspondence between **windows** and **access procedures**).



Using ***subsys_7*** as an example :

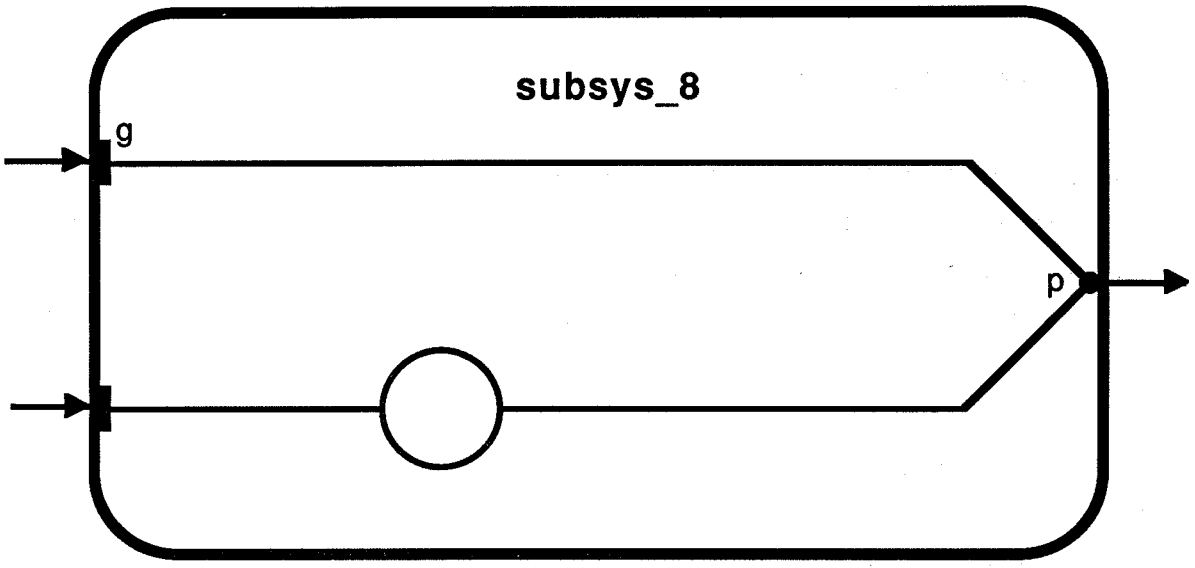
```
p3 = c.pw ;
g3 = c1.gw1
```

In the case of ***triplex***, all the externally visible **windows** are provided by **subsystems**.

```
out1 = dup_1.g1 ;
out2 = dup_2.g1 ;
out3 = dup_2.g2
```

To illustrate to boundary **window** to boundary **port** connection, a **path** straight through ***subsys_8*** (below) would be defined thus:

```
g = p
```



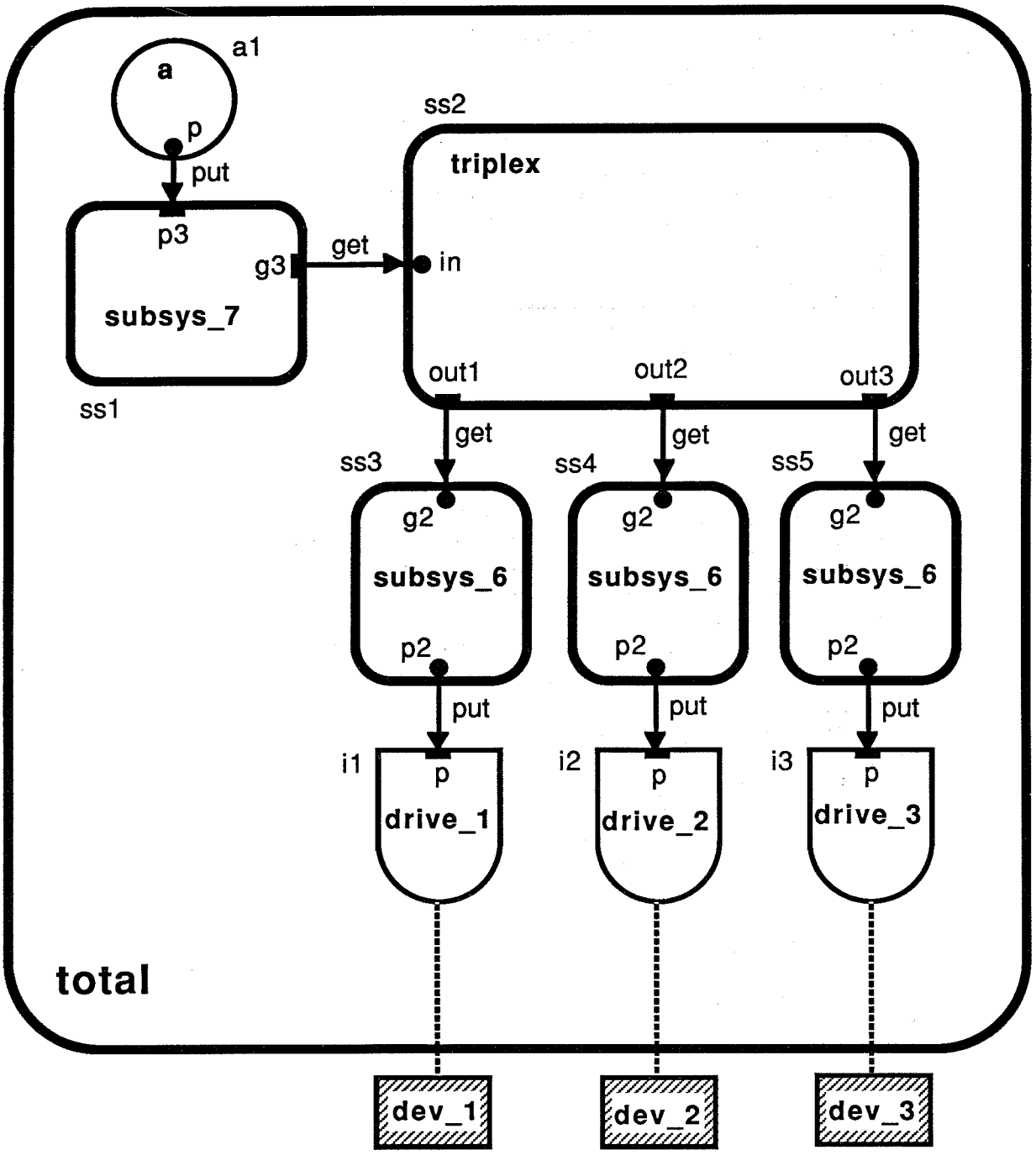
The following examples summarise the features of **subsystem modules**.

```
SUBSYSTEM subsys_5 ; { name part }
        { specification dependencies }
   PROVIDES g4 : get ;          { window specification }
   REQUIRES p4 : get ;
            c4 : com ;          { port specifications }
        { implementation dependencies }
   USES act, act_1, merge ;     { component templates }
        { components and interconnections }
   IDA m : merge ;
   ACTIVITY a : act (   p = p4,
                        tp = m.tw ) ;
   ACTIVITY a1 : act_1 (    c = c4,
                            tp1 = m.tw1 ) ;
        { equivalence list }
   g4 = m.g
END .
```

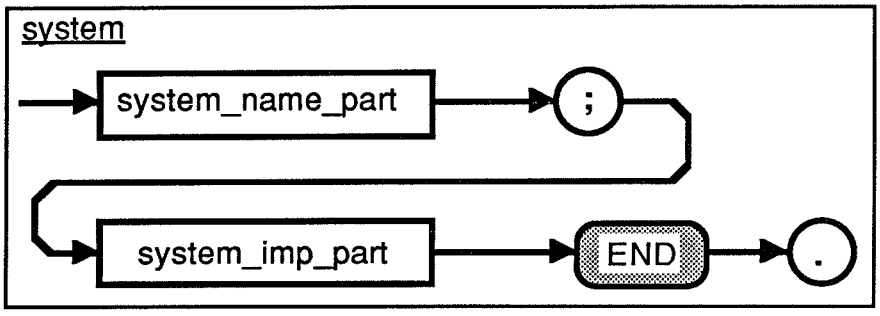
```
SUBSYSTEM subsys_6 ; { name part }
        { specification dependencies }
   { PROVIDES ..... ;           no window specifications }
   REQUIRES g2 : get ;
            p2 : put ;          { port specifications }
        { implementation dependencies }
   USES act_2, act_3, chan ;    { component templates }
        { components and interconnections }
   CHANNEL c : chan ;
   ACTIVITY a1 : act_2 (    gp = g2,
                            pp = c.pw ) ;
   ACTIVITY a2 : act_3 (    gp1 = c.gw,
                            pp1 = p2 ) ;
        { no equivalence list }
END .
```

```
SUBSYSTEM subsys_7 ; { name part }
        { specification  dependencies }
   PROVIDES p3 : put ;
                g3 : get ;          { window specifications }
   { REQUIRES ..... ;               no port specifications }
        { implementation  dependencies }
   USES act_4, ch, ch_1 ;     { component templates }
        { components and interconnections }
   CHANNEL c : ch ;
   CHANNEL c1 : ch_1;
   ACTIVITY a : act_4 (       gp = c.gw,
                              pp =  c1.pw1 );
        { equivalence list }
   p3 = c.pw ;
   g3 = c1.gw1
END .

SUBSYSTEM triplex ; { name part }
        { specification  dependencies }
   PROVIDES out1, out2, out3 : get ; { window specifications }
   REQUIRES in : get ;             { port specification }
        { implementation  dependencies }
   USES exp, dictionary,
        copy, duplicate ; { component templates }
        { components and interconnections }
   POOL look_up : dictionary ;
   SUBSYSTEM dup_1 : duplicate ;
   SUBSYSTEM dup_2 : duplicate ;
   ACTIVITY e : exp(  g = in,
                      ep = dup_1.p,
                      dp = look_up.dw) ;
   ACTIVITY c : copy( g = dup_1.g2,
                      cp =  dup_2.p);
        { equivalence list }
   out1 = dup_1.g1 ;
   out2 = dup_2.g1 ;
   out3 = dup_2.g2
END .
```

Finally in this section we turn to **systems**. As explained earlier, a **system** is simply a **subsystem** with no **ports** or **windows**. In the graphical form no **paths** cross the boundary of a **system** symbol which is otherwise identical to that representing a **subsystem**. The **system** diagram below uses three of the **subsystems** which have been discussed above.

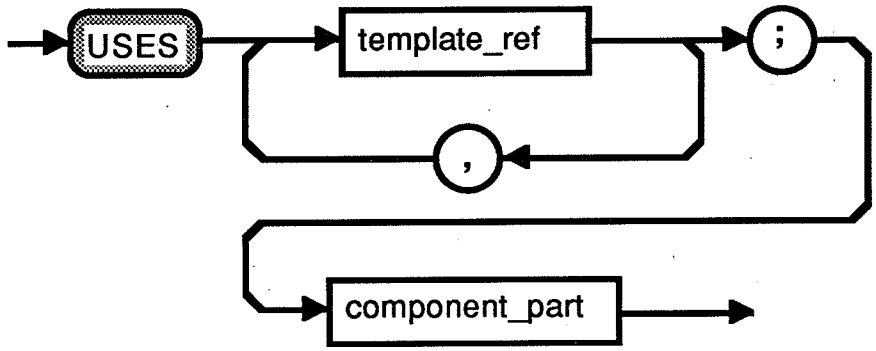


The syntax which describes the textual form of a **system** is a modification of that for a **subsystem** to take account of the absence of **ports** and **windows** and the fact that no ***equivalence list*** is required.



system_name_part

system_imp_part



The list of **templates** in the **USES** section may not include a **template** for a **system**.

Using **system** ***total*** as a basis, the following example summarises the features of a **system module**.

```
SYSTEM total ;                { name part }
        { implementation  dependencies }
   USES a, subsys_7, triplex, subsys_6, drive_1, drive_2, drive_3 ;
                                                  { component templates }
        { components and interconnections }
   SERVER i1 : drive_1 ;
   SERVER i2 : drive_2 ;
   SERVER i3 : drive_3 ;
   SUBSYSTEM ss1 : subsys_7 ;
   SUBSYSTEM ss2 : triplex ( in = ss1.g3 ) ;
   SUBSYSTEM ss3 : subsys_6 (   g2 = ss2.out1,
                                p2 = i1.p );
   SUBSYSTEM ss4 : subsys_6  (  g2 = ss2.out2,
                                p2 = i2.p );
   SUBSYSTEM ss5 : subsys_6 (   g2 = ss2.out3,
                                p2 = i3.p );
   ACTIVITY a1 : a ( p = ss1.p3 );
END .
```

2.5 ACTIVITIES

Description

Mascot application software is conceived in terms of a set of independent threads of execution which is mapped onto a hardware configuration containing an arbitrary number of processors. There are normally fewer processors than there are computational threads and it is the responsibility of the scheduling function of the Mascot **Context** software, acting either pre-emptively or in co-operation with the application software, to allocate the available processing power appropriately. The unit of scheduling for this purpose, an individual process, is known as an **activity**.

Thus **activities** are the fundamental processing elements in a Mascot application. Conceptually, all **activities** are executed in parallel. In practice, when any two communicating **activities** are considered, there may be literal parallelism resulting from their being mapped onto distinct processors or pseudo-parallelism if they share the same processor. In the latter case, depending on the mode of scheduling being employed, the synchronisation problems arising from their use of common data may be identical with those met with in true parallelism.

It is a principle of the Mascot philosophy that the mechanisms necessary to safeguard the integrity of data communicated between **activities** should be embodied not in the **activities** themselves but in a separate communication element. This is the intercommunication data area or **IDA** which is discussed in detail in a separate section. It encapsulates the shared data area and normally provides access to it through a procedural interface. Mascot **activities** therefore never communicate with each other directly but always through the intermediary of an **IDA**. In such transactions the **activity** is the active participant which invokes the operations specified in an **access interface** and provided by the passive **IDA.**

A consequence of this approach is that an **activity**'s influence on the remainder of the application software is restricted to its interactions with the **IDA**s to which it is connected. In particular, it is uninfluenced by the existence of any other **component**. Its part in the overall application is limited to using, processing and transmitting the data which flows to it along the **paths** of the **network**.

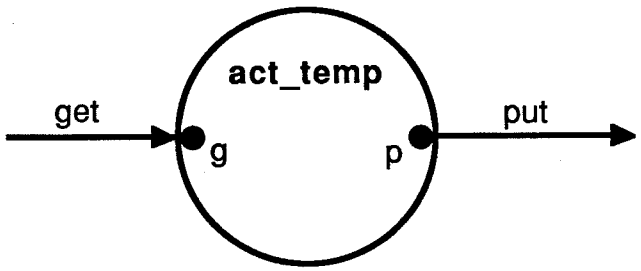
A Mascot **system** consists of a network of **activities** and **IDA**s. Each of the **activities** present has been created from an **activity template**. In some cases several of the **activities** may have been created from the same **template**. Since each **template** is a distinct design entity, developed independently, it is important that there are means of exercising control over the formation of a **network**. The validity of the connections must be capable of being checked during the construction process. The formal arrangements whereby a **Mascot database** records the **status** or degree of validation of each **module** is described in Section 3.1.

The validation is carried out in terms of **access interfaces**. An **activity template** establishes a **network** connection point by specifying a **port**. Each **port** implies a connection to a **path** of a particular type whose associated **access interface** specifies a set of data transfer or other operations. The coding necessary to implement these operations is provided by an **IDA** possessing a **window** of matching type. Valid **activity** to **IDA** connections are ones for which this type checking proves successful.

Graphical Representation

An **activity** is represented in a Mascot **network** diagram by a closed curve with rounded corners and drawn with a thin line. Normally, in accordance with historical convention, the curve is a circle. External network dependencies are shown by lines which pass through the boundary of the symbol and terminate, just inside the **activity** boundary, in the **port** symbol.

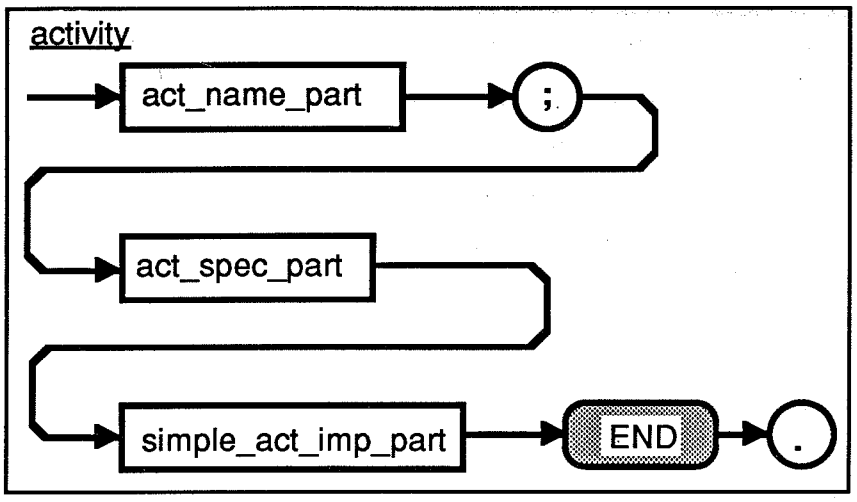
The diagram below shows an **activity template**. The name of the **template**, and of the **module** which represents it, is ***act_temp*** and this identifier is placed inside the **activity** symbol. As a general rule the name of an **activity** should clearly reflect its function. The **activity** possesses two **ports**, ***g*** and ***p*** and, as the labels on the two connected **paths** indicate, the external requirements which they represent are specified by **access interfaces** ***get*** and ***put***, respectively.



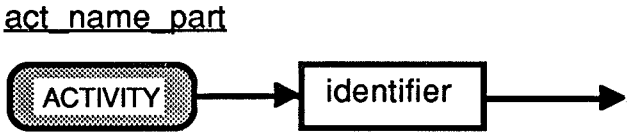
An **activity component**, as opposed to an **activity template**, would be represented as part of a **network** . In this case the symbol would represent a particular **component** created from a **template** and would be given a name distinct from that of the **template** itself. This name is placed outside the **activity** symbol. Examples are to be found in Section 2.4.

Textual Representation

In the mandatory subset of the Mascot definition, an **activity module** takes the syntactic form shown, in outline, below.



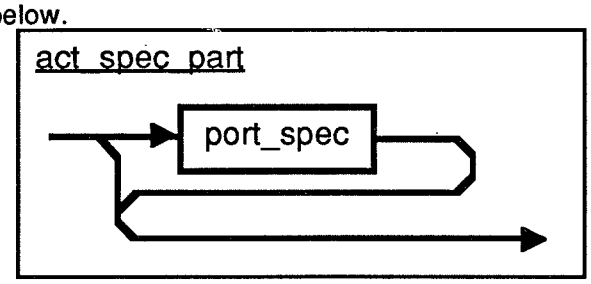
The ***name part*** establishes the **class** of the **module** and gives the **template** a name.



For example:

ACTIVITY act_temp ;

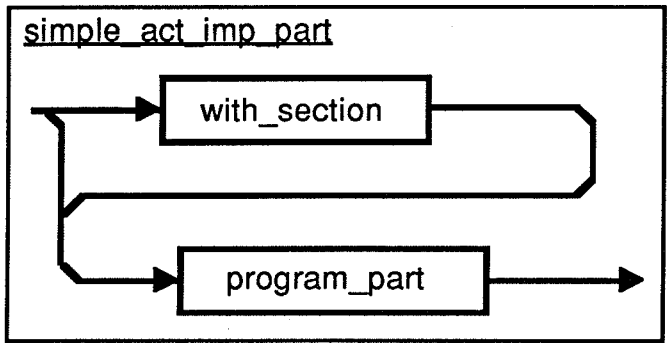
The ***specification part*** specifies the **network** dependencies. Its syntax is defined in the diagram below.



The form of **port** specifications is defined in a preceding section of the Handbook:

REQUIRES p : put ; g : get ;

Finally, the ***implementation part*** defines, explicitly or implicitly, the coding of this parallel thread of execution. Its syntax is shown in Pascal style below:



In its simplest form it consists solely of a block representing a set of private declarations and a sequence of program statements. The ***program part*** may define procedures, variables and constants etc. which are local to the **activity**.

Definitions of globally used data-types and symbolic constants are normally specified in **definition modules** (described in Section 2.3). They are made available for use in an **activity module** through the **WITH** section of its ***implementation part***. This simply lists the names of the relevant **specifications.**

```
WITH dataflow_types, real_constants ;
```

The following outline example summarises the features of **activity templates** described above:

```
ACTIVITY act_temp ;  { name part }
        { specification dependencies  }
   REQUIRES p : put ;
                g : get ;  { port specifications }
        { implementation dependencies  }
   WITH dataflow_types, real_constants ; { global definitions }
        { local declarations }

        { activity coding with initial entry point }

END .
```

2.6 INTERCOMMUNICATION DATA AREAS (IDAs)

Description

Any scheme for the organisation of co-operating parallel processes must include mechanisms for the control of data intercommunication. Where these mechanisms are imposed, globally, by a run-time system they may often present themselves as a constraint on the software design process. The Mascot philosophy is to provide the designer with the means to create precisely the communication facilities that are required between any two **activities** in the system. Previous sections of this Definition have described how the external network dependencies of an **activity** are expressed in terms of the procedure specifications which constitute the body of an **access interface**. These procedures, which perform the necessary data transfers, are implemented in a Mascot design element called an intercommunication data area or, more shortly, an **IDA**. Wherever information is transmitted between **activities** it is communicated through an intervening **IDA**; wherever common information is shared by **activities** it is stored in an **IDA**. Thus, in general, **IDAs** encapsulate both the shared data and the custom designed access mechanisms necessary to safeguard the integrity of information being held in or propagated through the **IDA**.

An **IDA** is a passive element. The code which it contains is never scheduled for execution in its own right. The independent threads of execution which form the run-time manifestation of **activities** simply pass through the **IDA** coding at points where intercommunication is taking place. Several such threads may simultaneously be active, or temporarily suspended, within an **IDA** which thus encapsulates the critical occasions where **activities** require concurrent access to the same data structure. When these circumstances arise they may involve more than one **activity** utilising the same access mechanism, and therefore actively using the same section of **IDA** code, or alternatively, several different access mechanisms may be operating concurrently on the same items of data. The Mascot method, while leaving the software designer firmly in control of the means of solving the system's concurrency problems, restricts the location of the coded solutions to this one type of component.

An **IDA**, therefore, is an encapsulated data type whose detailed physical representation is hidden from its users and whose component values may be manipulated indirectly through a procedural interface. It fulfils two principal purposes in a Mascot network. By providing mutual exclusion wherever necessary between **activities** competing for the use of a common resource, it safeguards the integrity of data. By providing cross stimulation between co-operating **activities**, it maintains the propogation of data in the network.

Through experience with earlier versions of Mascot, two simple classes of **IDA** have been found to be particularly useful. The first of these provides for uni-directional transmission of data from one or more producer **activities** to one or more consumers. It is known as a **channel** and is characterised by a destructive read operation and a non-destructive write. As a consequence it can become empty of data and, as its capacity is finite, it can also become full. The other specially useful type of **IDA** is known as a

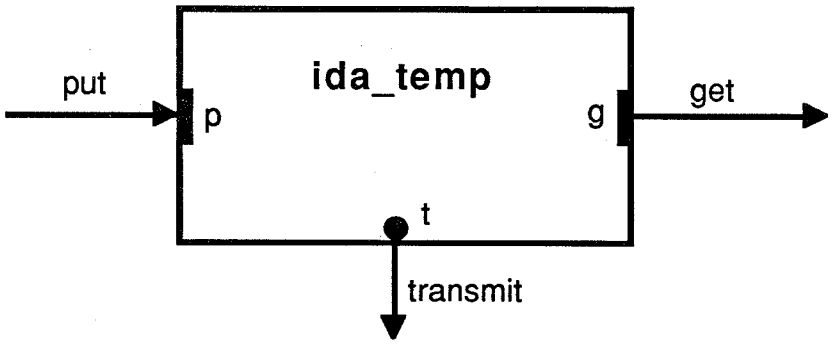
pool. In this case it is the write operation which is destructive and the read non-destructive. A **pool** is therefore a suitable kind of **IDA** to represent a table or dictionary which **activities** periodically consult or update. Many of the simpler Mascot networks can be built satisfactorily from **activities**, **channels** and **pools**. The graphical forms of such networks are normally referred to as ACP (Activity-Channel-Pool) diagrams.

In general, active and passive components, **activities** and **IDAs**, alternate along the **paths** of a Mascot network. Certainly adjacent activities never occur. However, it is sometimes necessary for data to be projected directly from one **IDA** to another. Typically this requirement might arise where the two **IDAs** are to be accommodated in separate storage units addressable by separate processors. Mascot caters for this contingency by allowing **IDAs** to possess **ports** as well as **windows**. Thus an **IDA** may use access mechanisms as well as providing them and so may be connected, by a **path**, to another **IDA**.

Graphical Representation

An **IDA**, in its most general form, is represented in a Mascot **network** diagram by a rectangular shape. The square corners of the symbol indicate the passive nature of the component it represents. Throughout the Mascot graphical notation, an exclusively square cornered symbolic outline certifies that decomposition, to any depth, will not reveal an active element. A rounded corner, on the other hand, affirms the probability, but not the certainty, of one or more independently scheduled threads of execution being involved.

External network dependencies are denoted by lines which pass through the boundary of the symbol. These lines are terminated, within the boundary, by a **window** symbol. The diagram below shows an **IDA template**.

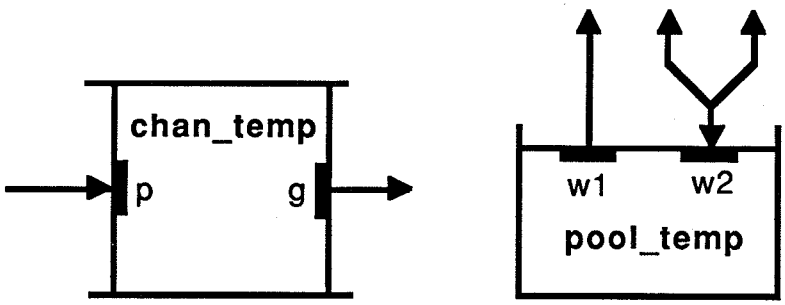


The name of the **template** is ***ida_temp*** and this identifier is placed inside the rectangular **IDA** symbol. As a general rule the name of the **template** should clearly reflect its function. The **IDA** possesses two **windows**, ***p*** and ***g***, represented by the filled rectangular symbols with which two of the externally connecting **paths** terminate. The labels on these **paths** show that the procedures provided by the **IDA** at the corresponding **windows** are specified in the **access interfaces**, ***put*** and ***get***, respectively.

A third **path**, labelled ***transmit***, is shown crossing the bottom boundary of the symbol and this terminates in the **port**, *t*. The procedures specified in the corresponding **access interface**, ***transmit***, may be used by the **IDA** coding in order to project information directly to or from another **IDA**. This mode of data transfer takes place only when a thread of execution passes through the **IDA**.

An **IDA component**, as opposed to an **IDA template**, would be represented as part of a **network**. In this case the symbol would represent a particular **component** created from a **template** and would be given a name distinct from that of the **template** itself. This name is placed outside the **IDA** symbol. Examples are to be found in Section 2.4.

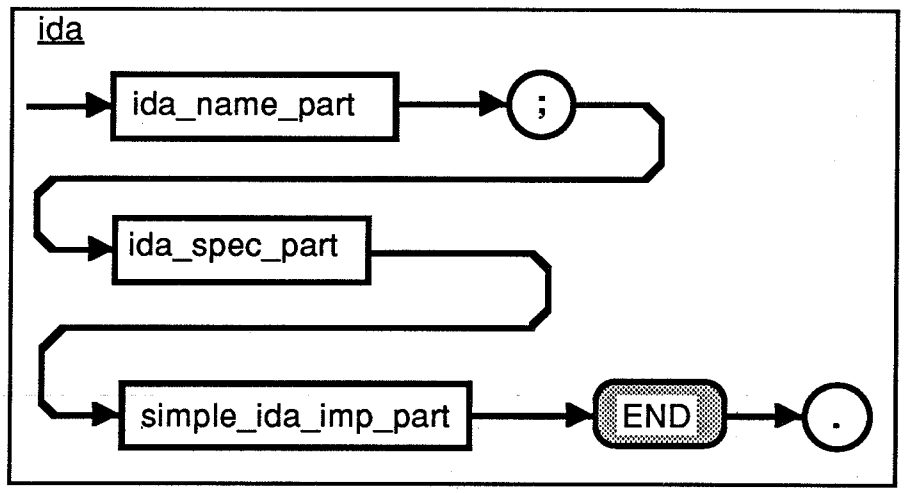
Two alternative graphical representations are available for **channels** and **pools** (described earlier) and these are shown below.



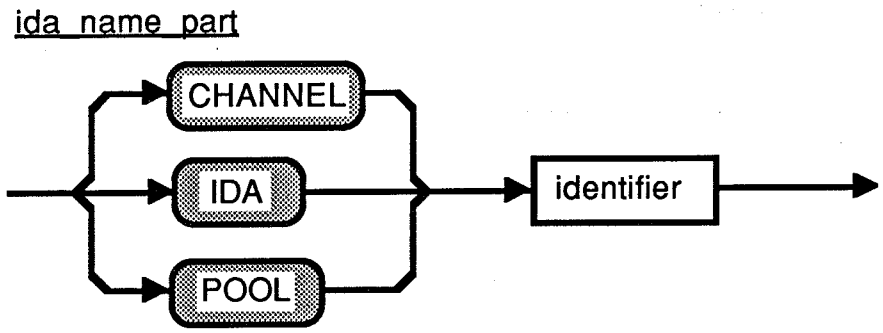
The standard rectangular symbol is slightly modified in each case to make it more reminiscent of the original Mascot 2 **channel** and **pool** symbols.

Textual Representation

In the mandatory subset of the Mascot definition, an **IDA module** takes the syntactic form shown, in outline, below.



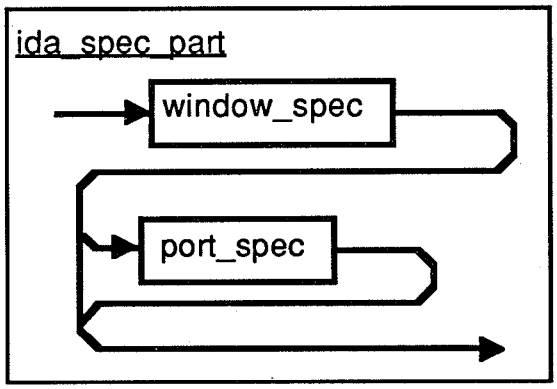
The ***name part*** establishes the **class** of the **module** and gives the **template** a name. It also distinguishes between a generalised **IDA** and the two special cases: **channel** and **pool**.



Examples are:

```
IDA ida_temp ;
CHANNEL chan_temp;
POOL pool_temp;
```

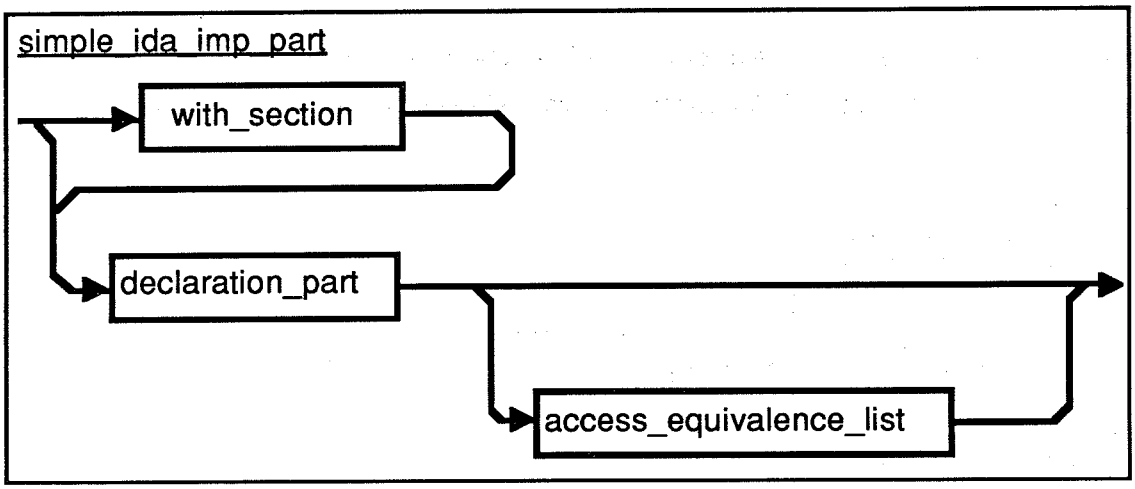
The ***specification part*** specifies the **network** dependencies. Its syntax is defined in the diagram below.



The form of the **window** and **port** specifications is defined in a preceding section of this Handbook. Notice that at least one **window** must be specified.

```
PROVIDES p : put ;
         g : get ;
REQUIRES t : transmit ;
```

The ***implementation part*** defines, explicitly or implicitly, all the coding contained in the **element**. Its syntactic structure is defined below.



In its simplest form it consists solely of a ***declaration part***.. Program objects to which external access is provided, via a **window** of the **IDA**, are distinguished from purely local entities by the use of the word **ACCESS** in their declarations. Ignoring, for the moment, the WITH section an elementary example might begin:

```
CONST
   bufsize = 100 ;
VAR
   buffer : ARRAY[1 .. bufsize] OF data_flow_type ;
   in_pointer, out_pointer : integer ;
   inq, outq : controlq ;
ACCESS PROCEDURE put_data ( item : data_flow_type ) ;

ACCESS FUNCTION get_data : data_flow_type ;
```

In this **IDA**, a data buffer is declared together with a pair of pointers and two control variables, ***inq*** and ***outq***, whose significance in the Mascot method of process synchronisation is explained in the relevant section of this Handbook. In conformity with the principle of data hiding, all these variables are entirely private to the **IDA**. The procedure declaration section must include an implementation of every procedure made externally available in the **IDA**'s **PROVIDES** list. Thus if this was:

```
PROVIDES p : put ; g : get ;
```

then, **access interface *put*** might contain the specification :

```
PROCEDURE put_data ( i : data_ flow_type ) ;
```

and **access interface *get***, the specification :

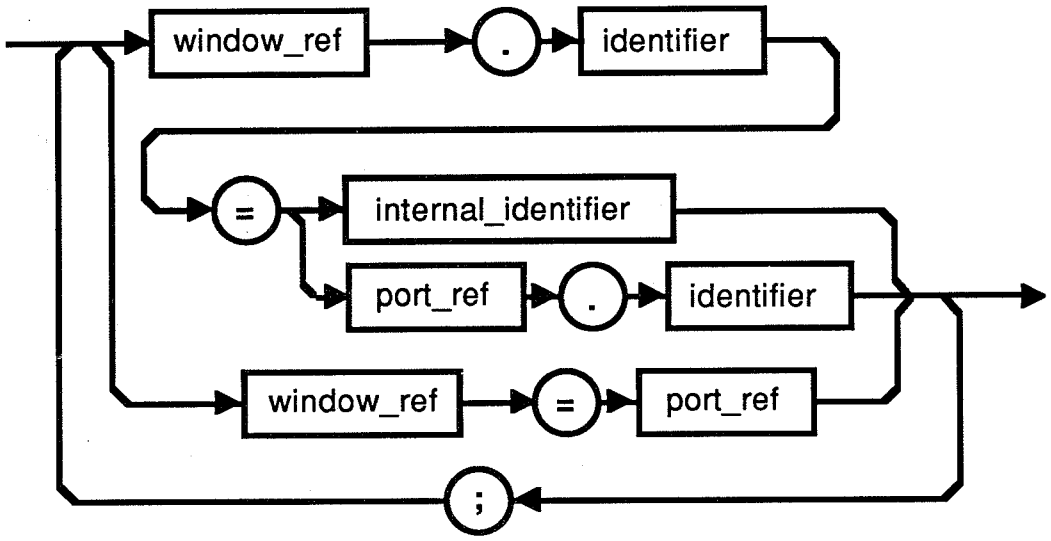
```
FUNCTION get_data : data flow_ flow_type ;
```

The purpose of the **window** identifiers, ***p*** and ***g***, will be explained shortly.

Three other special procedures may be included in the ***declaration part*** and identified by the words **INITIALISATION**, **RESET** and **TERMINATION** respectively. Their execution is invoked by system control functions as described in Section 4.6 of the Handbook. Any other procedures declared here are local to the **IDA**. They can only be executed as a result of being called from either ***get_data*** or ***put_data*** . The procedures ***get_data*** and ***put_data*** themselves are of course invoked from **activities** connected by **paths** to the appropriate **window**. They constitute the mechanism whereby access is obtained to the hidden data buffer.

There is a need to establish the correspondence between the access mechanisms in an **IDA** and the procedures which the **windows** indicate are to be provided. This may be achieved by identity of names as in the above example, or alternatively by the optional ***access equivalence list*** shown in the ***implementation part*** of the syntax.

access_equivalence_list



Thus, there is provision for equating each identifier in each **window** with a corresponding identifier declared inside the **IDA**; this is known as window-to-local equivalence. In addition where, for example, data propagation takes place directly between two connected **IDA**s, an identifier in a **window** may be equated to an identifier in a **port** so that it is a second **IDA** which provides the functionality; this is known as window-to-remote equivalence. For the particular case where all the functionality of a **window** is to be provided externally there is a shorthand form which equates all the features of the **window** to those of a matching **port** in a single access equivalence statement; this is known as window-to-port equivalence.

Two examples of window-to-local equivalence are:

```
p.put_data = put_data ;
g.get_data = get_data
```

In general, using this notation, the corresponding procedure names need not be the same. Indeed, like the names of corresponding formal and actual procedure parameters, they may be expected to be different in other than simple instances of **IDAs**.

If our example **IDA** possesses a **port**:

```
REQUIRES t : put ;
```

it could be equated with the matching **window**:

```
p = t
```

implying that the the **access interface** procedure ***put_data*** is to be provided from elsewhere. Alternatively, an individual procedure in the **window** may be equated to a procedure in the **port**:

```
p.put_data = t.put_data
```

It will be observed that, in the above example, data-types are used which are not defined in the **IDA**. This will normally be the case with data-types and symbolic constants which are used in more than one **module**. In order to supply such global definitions, a **specification module**, known as a **definition** and described in Section 2.3 of the Handbook, is provided for them. It is the purpose of the **WITH** section in the ***implementation part*** of the simple **IDA** to import such additional global definitions that are needed but which are not inherited from an **interface**.

```
WITH data_type_defs, control_type_defs ;
```

The following outline example summarises the features of **simple IDAs** described above :

```
IDA ida_temp ; { name part }
        { specification dependencies }
   PROVIDES p : put ;
            g : get ; { window specifications }
   REQUIRES t : transmit ; { port specification }
        { implementation dependencies }
   WITH ..... ;    { global definitions }
        { local declarations }

   ACCESS PROCEDURE ..... ;     { to match all procedures }
                                { specified }
   ACCESS FUNCTION ..... ;      { in the windows provided }
        { access equivalence list }
END .
```

2.7 SERVERS

Description

A Mascot application interacts with its environment via a set of peripheral devices attached to the processor or processors on which it is running. These devices generally appear to the software as sources, sinks or temporary repositories of data. In some cases, however, they appear as initiators of signals, acquainting the software with information concerning external events. The range of processors for which Mascot systems are likely to be implemented possess a variety of different interrupt handling architectures. Absolute standardisation is therefore not possible and this section of the Definition is intended to establish guidelines as to what Mascot facilities are provided to allow devices to be handled in a manner appropriate to the application.

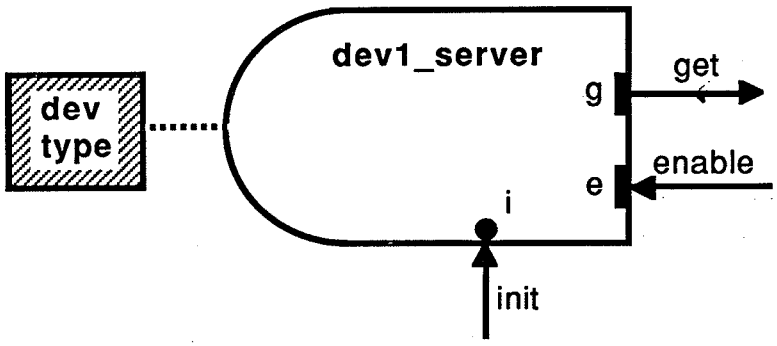
In order to enhance the portability and flexibility of a Mascot system, it is desirable that the application dependent details, relating to specific devices and to the architecture of particular computers, should be localised. This requirement is met by a Mascot design element called a **server** which encapsulates the mechanisms that control and transfer data to and from a particular device. **Servers** communicate with other Mascot components through **access interfaces** which enable them to be treated, as far as possible, like **IDAs**. The aim is to hide the application dependent details in the same way as the physical representation of shared data structures is hidden.

The constituent parts of a **server** vary according to the characteristics of the device with which it is concerned and the uses to be made of this device by the application software. The interactions it provides will normally include 'driver' mechanisms to allow control signals and/or data to be sent to the device. When an interrupt occurs, the action of the Mascot scheduling function is overridden by hardware and control is transferred to a section of code encapsulated in the **server** and known as a **handler**. It is the **handler** which transfers information between the, frequently volatile, data registers of the device and the internal data structures of the **server** which are accessible to the Mascot network via **windows**.

Servers share many of the attributes of **IDAs**; they may possess both **windows** and **ports**. They differ from **IDAs**, however, in being the only Mascot design elements which may include **handlers** and other code which communicates directly with devices. Where the mechanisms used for device interaction are not available to the application software but are provided by low level procedures within the environment, the use of these procedures is restricted to code encapsulated by a **server**.

Graphical Representation

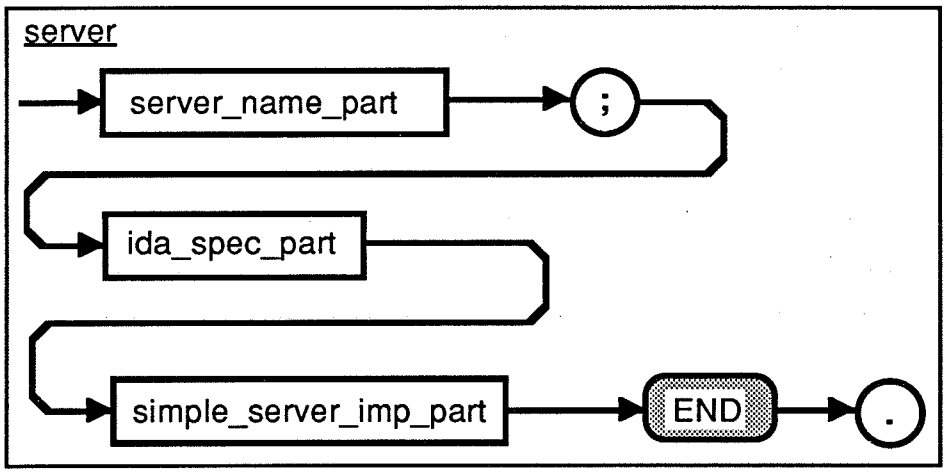
A **template** for a **server** is represented in a Mascot network diagram by a D-shaped symbol as illustrated below.

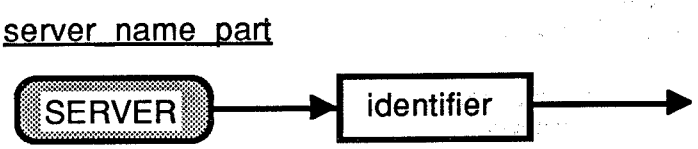


This **template** is called ***dev1_server*** and possesses two **windows**, ***g*** of type ***get*** and ***e*** of type ***enable***, and a single **port**, ***i*** of type ***init***. The various forms of connection are shown in their conventional positions. The rounded edge of the **server** symbol faces towards the device, **windows** are placed on the opposite, square cornered edge and **ports** on one of the sides of the symbol (top or bottom of the D). The orientation of the complete symbol may be chosen to suit the layout of the diagram. The diagram also shows the device controlled by the **server**. This is represented here by a hatched rectangle but a schematic sketch of the hardware would be equally acceptable. The presence of the device symbol and its connection to the **server** are optional.

Textual Representation

In the mandatory subset of the Mascot definition, a **server module** takes the syntactic form shown, in outline, below.



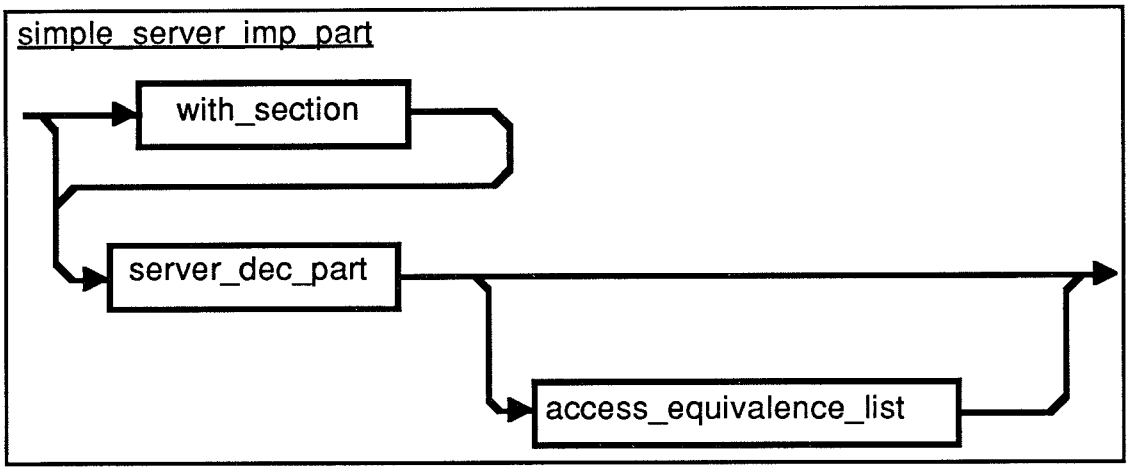


The ***name part*** identifies the **class** of the **module** and gives the **template** a name. For example:

SERVER dev1_server ;

The ***specification part*** is identical with that of an **IDA** as described in the previous section of the Definition. Reference to that section shows that **ports** may be included and that at least one **window** must be specified. Without the latter it could convey no information to the network and, therefore, perform no useful role.

The syntactic structure for the ***implementation part*** of a **server** is shown below.



First an optional **WITH** section allows globally accessible data type definitions to be imported from **definition specifications**. The explicit contents of the **server template** are contained in the succeeding ***declaration part***. . This is similar to that of an **IDA**. It shares with the **IDA** the requirement that its procedure declarations include implementations (distinguished by the word **ACCESS**) of the procedure headings offered by its one or more **windows**. Correspondence between the **access interface** headings and the internally declared procedures is established, as for **IDAs**, either by name identity or by means of an ***access equivalence list***, the form of which is described in the previous section of the Definition. **INITIALISATION**, **RESET** and **TERMINATION** procedures may also be included (see Section 4.6 of the Handbook).

Unique to this type of **template**, is the abilty to declare **handlers**. These routines are distinguished from other procedures by use of the design language word **HANDLER** in place of **PROCEDURE**. Now follows an outline example, based on the earlier graphical example, of a complete **module** representing a **server**.

```
SERVER dev1_server ; { name part }
        { specification dependencies }
   PROVIDES g : get ;
                e : enable ; { window specifications }
   REQUIRES i : init ;      { port specifications }
        { implementation dependencies }
   WITH ..... ;             { global definitions }
        { local declarations }
   ACCESS PROCEDURE ..... ;          { including all procedures }
                                     { specified }
   ACCESS FUNCTION ..... ;           { in the windows provided }
   HANDLER ..... ;    { handler declaration }
        { access equivalence list }
END .
```

2.8 TEMPLATE CONSTANTS

Description

This feature is not part of the mandatory subset of the Mascot definition.

Collections of **objects**, of any kind, although all created from the same **template**, may nevertheless be required to behave differently from each other in minor ways. These individual secondary characteristics may be bestowed by specifying, in the **template**, constants whose values are to be supplied to individual **components** of this type. Such constants constitute another form of external dependency and are known in Mascot as **template constants**.

The range of **template constant** types is dependent on the implementation language, but could be expected to include any of the implicit data types of the language. The use made of these values is again dependent on the implementation language but they could typically be used for:

- device addresses (in a **server**)
- interrupt levels (in a **server**)
- buffer sizes (any **simple template**)
- iteration control (any **simple template**)

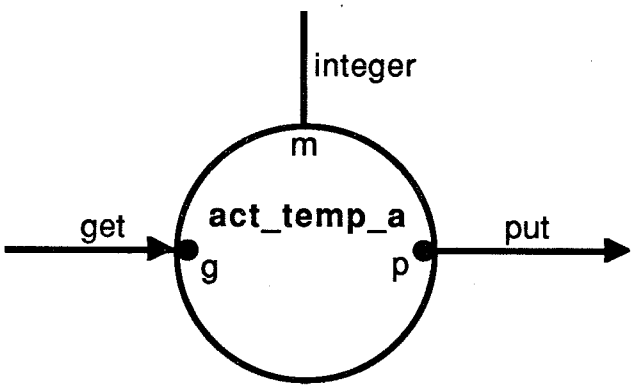
This latter use might for instance govern the number of terms in a series to be evaluated and hence the accuracy and computation time of a result.

Just as network **paths** may be carried, via **'port** to **port'** or **'window** to **window'** connections, across the boundaries of **composite** design elements, so **template constants** may be transmitted in a similar way to individual **components**. Indeed powerful use of the facility may be made by bringing **template constant** dependencies out to the enclosing **system** to be supplied dynamically during system **building** (see Section 3.2 of the Handbook).

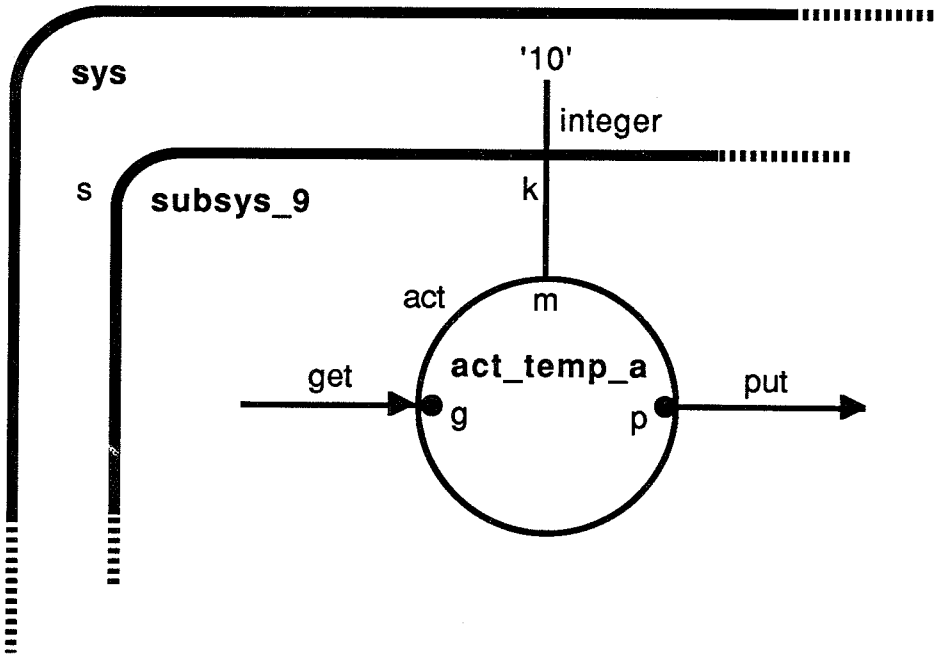
Template constants may be specified either individually or in the form of arrays in any **template**.

Graphical Representation

A **template** constant is distinguished from a communication **path**, in a Mascot network diagram, by the absence of any terminating symbol (such as those representing **ports** and **windows**) inside the boundary of the **template** or **component** to which it belongs. Taking an **activity template** as an example:



The **template** ***act_temp_a*** possesses a **template constant**, of type integer, known internally as ***m***. Supposing ***act_temp_a*** to be used to generate a **component** of a surrounding **subsystem** whose instantiation is to supply the value of ***m*** , this would be indicated graphically as shown below:

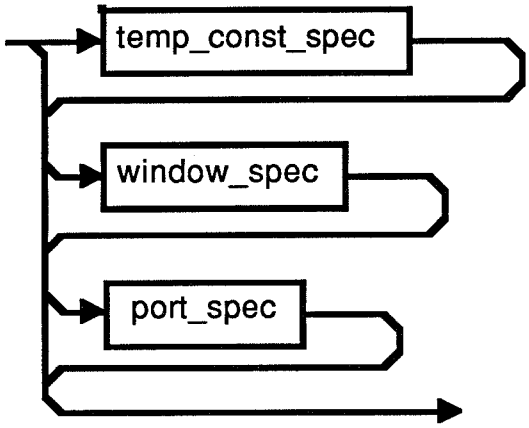


This diagram shows that the **template constant**, which is to be known as ***k*** in the **subsystem**, is to be given the value ***10*** when the subsystem is itself created. Similar diagrammatic conventions apply to all other types of **template**.

Textual Representation

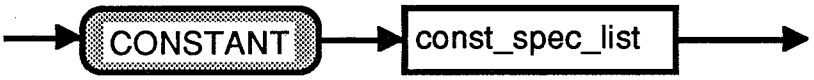
Reference to the complete syntactic descriptions (to be found in Appendix A) of any of the Mascot **templates** shows that **template constants** may be specified at the beginning of their ***specification parts*** . In the case of a **subsystem**, for example, the complete syntactic structure of its ***specification part*** (as opposed to that previously presented for the mandatory subset) is:

subsys_spec_part

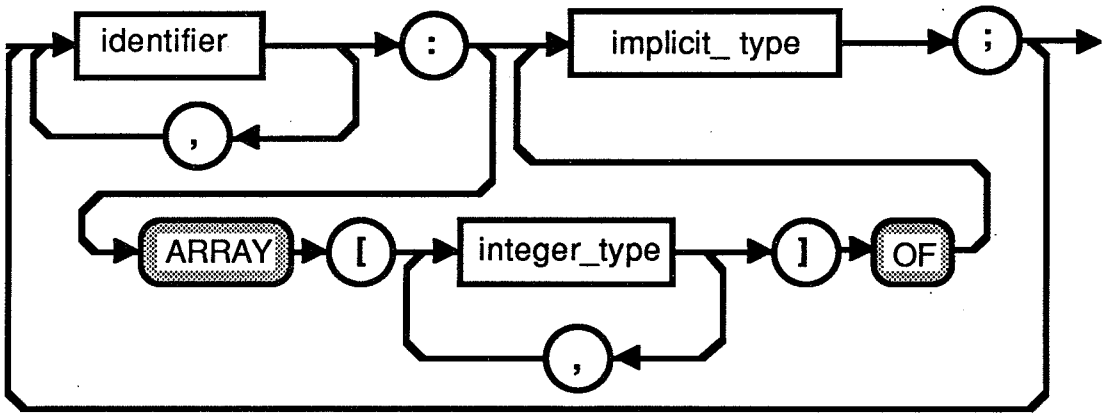


The additional syntactic structure is shown below:

temp_const_spec



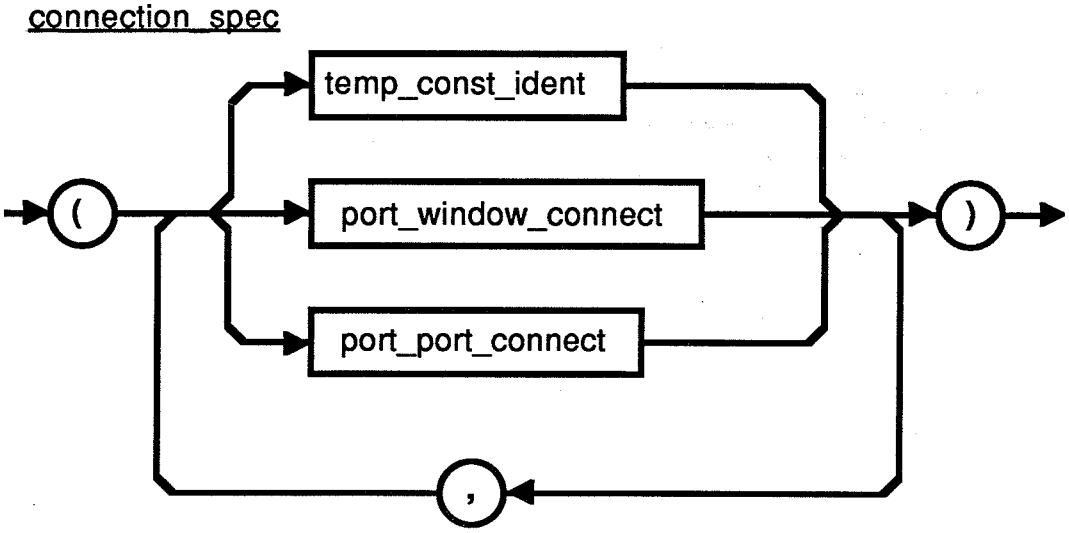
const_spec_list



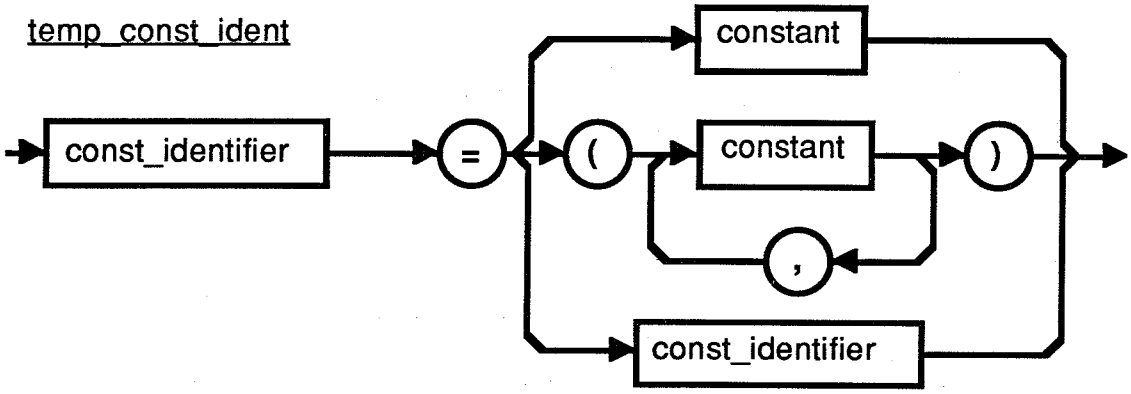
Using the graphical example above to illustrate the textual form, the **activity template** would include:

```
ACTIVITY act_temp_a ;
   CONSTANT m : integer ;

END.
```

The enclosing **subsystem** transmits the constant through the connection specification of the **component** derived from **template** ***act_temp_a*** . This is the same notation as that used to provide **port** to **window** and **port** to **port** connections. The complete syntax for a ***connection_spec*** is:



The additional structure required to identify **template constants** is:



This identifies a **template constant** with either its ultimate value or with another constant identifier at a higher level. Thus, the **subsystem** in our example would contain:

```
SUBSYSTEM subsys_9 ;
   CONSTANT k : integer ;

   ACTIVITY act : act_temp(    m = k ;
                               p = .... ;
                               g = .... ) ;

END.
```

and the **system**:

```
SYSTEM sys ;

   SUBSYSTEM s : subsys_9 ( k = 10 ; ..... ) ;

END .
```

Finally, an array of **template constants** might be specified as:

```
CONSTANT c : ARRAY[1..10] OF char ;
```

and its values provided by a construction similar to an Ada positional aggregate:

```
c = ('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j' )
```

Alternatively, the array name ***c*** may be equated with that of an assignment compatible array at a higher level of the design hierarchy.

2.9 LIBRARIES

Description

This feature is not part of the mandatory subset of the Mascot definition.

In Section 2.12 of the Handbook it will be seen that facilities exist in Mascot for the sequential decomposition of **activities**. The products of decomposition are **roots** and **subroots** whose relationships to each other, and to the adjacent parts of the data flow network, are formally shown on the diagram. This feature is limited to **activities** but all **simple templates** are open to procedural decomposition in terms of **libraries** which may be shared by any number of **templates**. A **library**, which may be instantiated in any **composite module**, consists of a set of externally accessible procedures encapsulated with other purely private declarations. It does not contain static data and consequently cannot be used as an **IDA**.

A Mascot **library** implements one or more **library interfaces** specifying which of the program objects declared in a given **library** may be used by **templates** possessing a **LIBRARY** statement which refers to that **interface**. Candidates for including such a specification are **simple IDAs**, **libraries**, **simple activities** and the **simple components** of **composite activities**. It is clear, therefore, that **libraries**, like **IDAs**, must be capable of supporting multi-threaded operation. However they must not allow interaction between the threads and so are not involved in problems of process synchronisation.

Graphical Representation

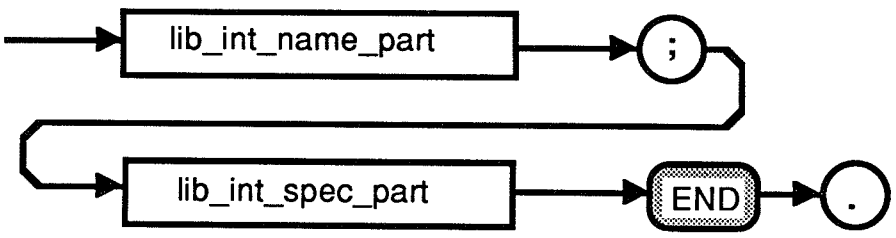
Neither **libraries** nor **library interfaces** are represented on Mascot network diagrams.

Textual Representation

Library Interface

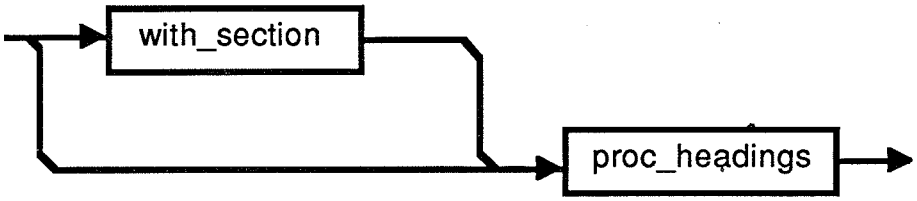
A **library interface specification** has the following outline syntax:

library_interface



lib_int_name_part

lib_int_spec_part



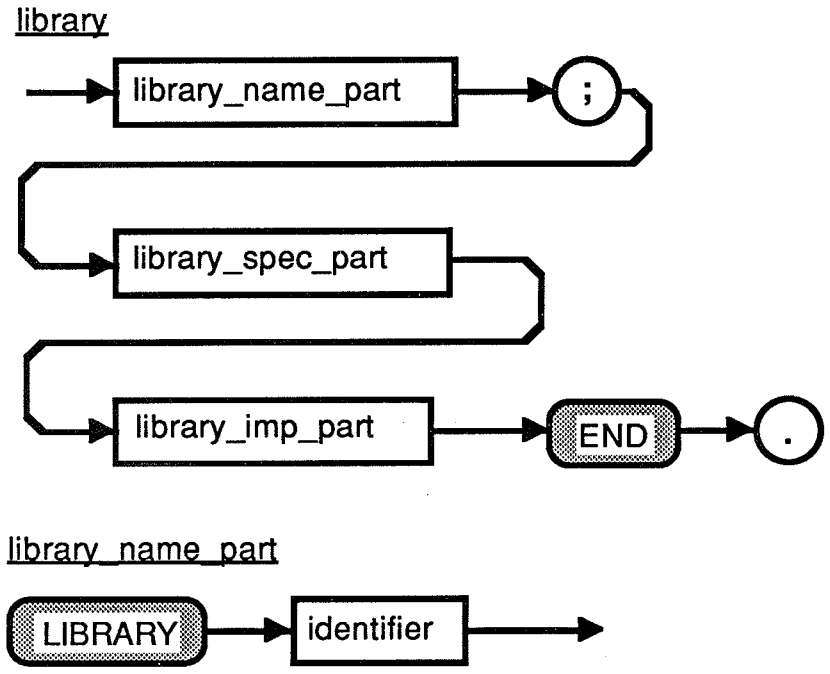
The optional **WITH** section allows global type and symbolic constant definitions to be imported to the **library interface**.

Notice that variables are not allowed in a **library interface**. Some examples follow:

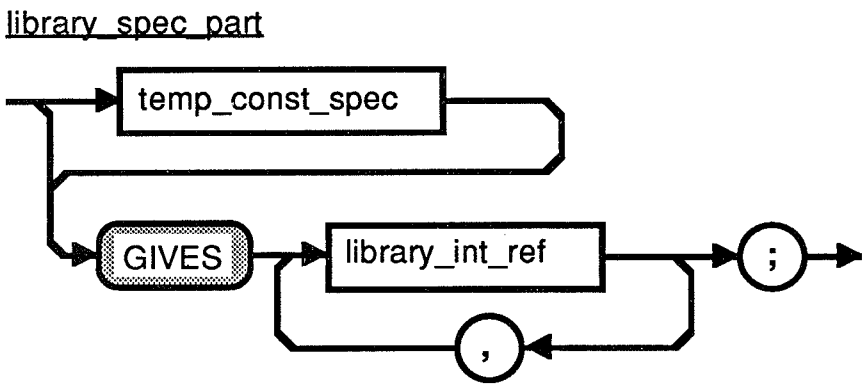
```
LIBRARY INTERFACE trig_functions ;
   FUNCTION sin ( x : real ) : real ;
   FUNCTION cos ( x : real ) : real ;
   FUNCTION tan ( x : real ) : real ;
END .

DEFINITION complex ;
   TYPE
      complex = RECORD
                   real_part : real ;
                   imaginary_part : real
                END ;
END .

LIBRARY INTERFACE complex_1 ;
WITH complex ;
   FUNCTION add ( x, y : complex ) : complex ;
   FUNCTION sub ( x, y : complex ) : complex ;
END .


LIBRARY INTERFACE complex_2 ;
WITH complex ;
   FUNCTION div ( x, y : complex ) : complex ;
   FUNCTION mult  x, y : complex ) : complex ;
END .
```

Libraries

The outline syntactic structure of a **library module** is shown below



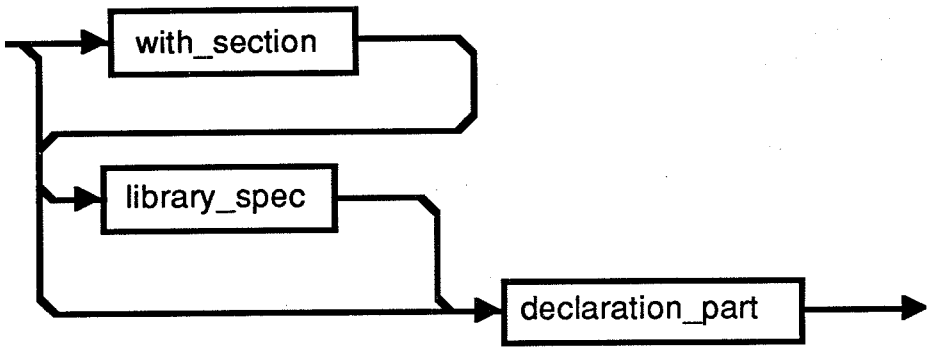
The ***specification part*** permits the specification of **template** constants and requires that the **library interface**, for which this **template** describes an implementation, should be identified in a **GIVES** section.



Where a **library** is to give more than one **Interface**, the interactions provided in the **specifications** must be distinguishable from each other.

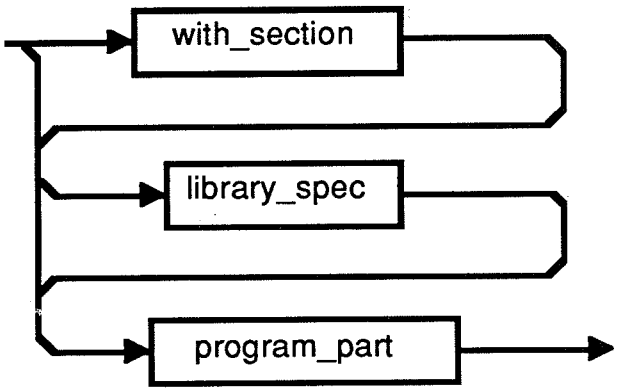
It can be seen from the syntax of the ***inplementation part*** , given below, that **definition** and **library** dependencies may be included

library_imp_part



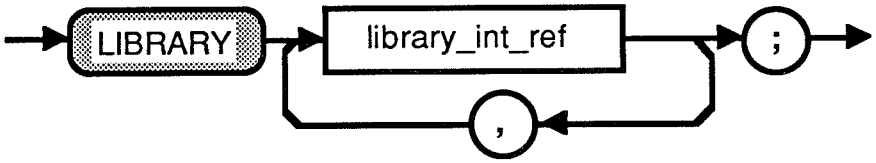
A ***library specification*** may appear in any **activity, IDA, server** or **library template**. Thus in the complete Mascot definition, an **activity** ***implementation part*** takes the form in Pascal style:

simple_act_imp_part



The additional structure is a list of **library interfaces**:

library_spec

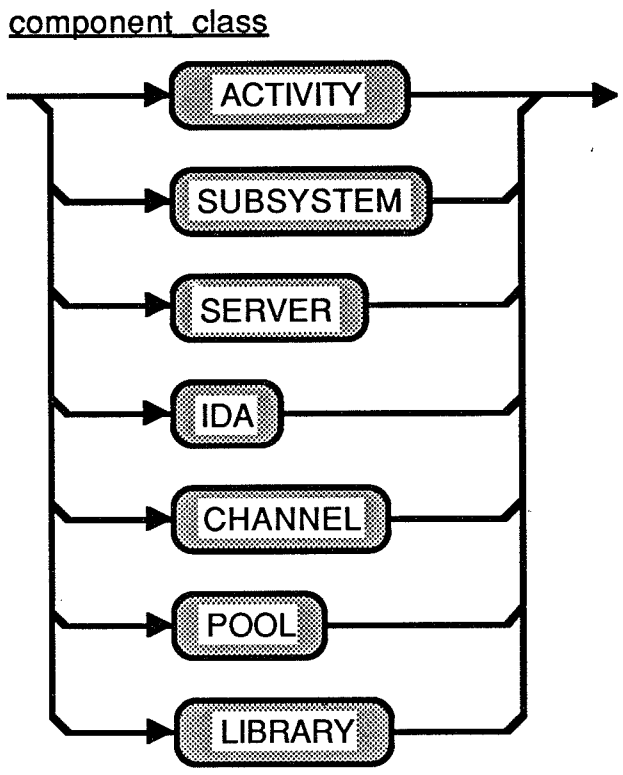


The remainder of the **library** ***implementation part*** takes the form of a set of declarations which may not, however, include variables. This reflects the fact that static variables are not allowed in a **library**. In some implementation languages this prohibition might be expressed in a different manner. The procedure declarations must include all the procedures specified in the given **interfaces**. The correspondence is by procedure name.

The following is an example of a **library template**.

```
LIBRARY trig_lib ;
        { specification dependencies }
{ CONSTANT ..... ;   no template constants }
GIVES trig_functions ;
        { implementation dependencies }
{ WITH ..... ;          no definition dependencies }
{ LIBRARY ..... ;       no library dependencies }
        { local declarations }
   PROCEDURE .... ;  { including all procedures specified }
   FUNCTION ..... ;  { in library interface trig_functions }
END .
```

Finally, a **library module** may be instantiated within any **composite module** and then becomes implicitly available throughout all the other **components**. Thus the full syntax for the the ***component_class*** in the ***implementation part*** of a **network module** (given in part in Section 2.4) is:



Taking **subsystem** ***subsys_6*** (Section 2.4) as an example, **library** ***trig_lib*** could be made available to the two **activities**, ***a1*** and ***a2***, and the **IDA** ***c*** by extending the **subsystem module** as shown below:

```
SUBSYSTEM subsys_6 ; { name part }
        { specification  dependencies }
   { PROVIDES ..... ;          no window specifications }
   REQUIRES g2 : get ;
            p2 : put ;       { port specifications }
        { implementation  dependencies }
   USES act_2, act_3, chan, trig_lib ; { component templates }
        { components and interconnections }
   LIBRARY t : trig_lib;
   CHANNEL c : chan ;
   ACTIVITY a1 : act_2 (   gp = g2,
                           pp = c.pw ) ;
   ACTIVITY a2 : act_3 (   gp1 = c.gw,
                           pp1 = p2 ) ;
        { no equivalence list }
END .
```

The **library template** name has been added to the **USES** statement and a **library component**, derived from this **template**, defined in the **module**. An instance name for the new **component** is included for the sake of consistency but is not used for any practical purpose.

Use of the **library interface**, ***trig_functions***, implemented by ***trig_lib***, would be indicated in the **template** ***act_2*** as follows:

```
ACTIVITY act_2 ; { name part }
        { specification  dependencies }
   REQUIRES gp : get ;
            pp : to ;  { port specifications }
        { implementation  dependencies }
   WITH ..... ;   { global definitions }
   LIBRARY trig_functions ;  { library interfaces used }
        { local declarations }
   END
END .
```

The coding of the **activity** could then call the functions ***sin***, ***cos*** and ***tan***.

Libraries may also be instantiated in **modules** which represent sequentially **composite** design entities rather than **networks** (see Section 2.12).

2.10 COMPOSITE IDAs

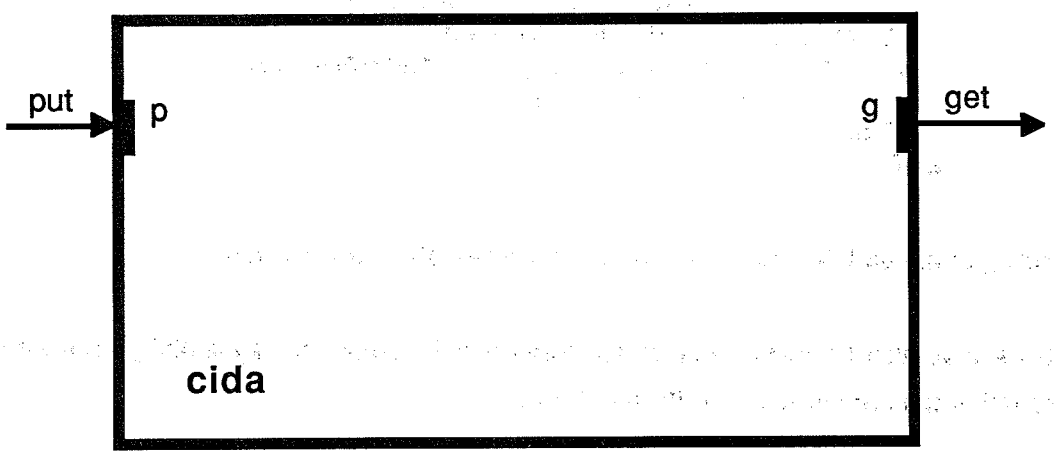
Description

This feature is not part of the mandatory subset of the Mascot definition.

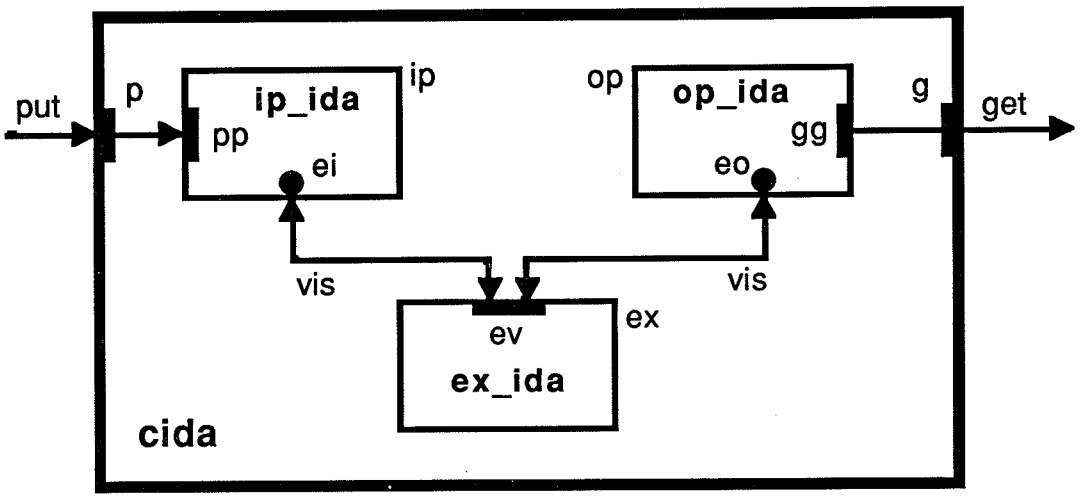
Often, where a network of interconnected **IDAs** occurs in a Mascot design, it takes the form of a **composite component**. Such **composite IDAs** furnish another means of performing network decomposition in Mascot, supplementing that described in Section 2.4. A **composite IDA** consists of a network of internal **IDAs** connected by **paths** and thus utilises the facility, described in Section 2.6, whereby **IDAs** may possess **ports**.

Graphical Representation

Composite IDAs are represented in higher level network diagrams by symbols of the same shape as those used for **simple IDAs**. Their possible external dependencies, **paths** terminating in either **windows** or **ports**, are also represented in an identical manner. The following diagram illustrates a **template** for a **composite IDA**. Its **composite** nature is indicated by the use of a thick line for its boundary. Alternatively a double line would have the same significance and other conventions, defined for some specific set of documents, would be acceptable.



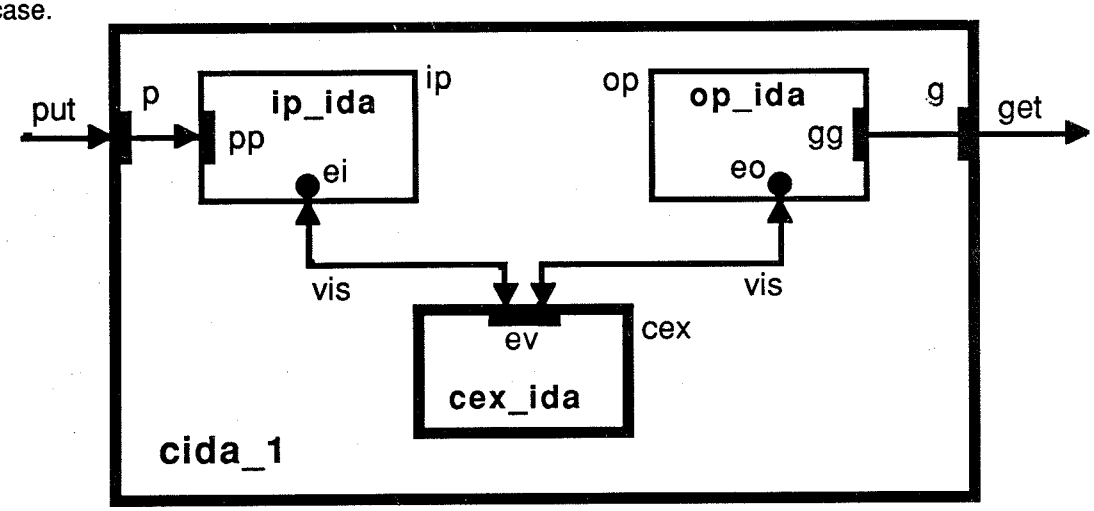
The squared corners proclaim that this symbol represents a wholly passive entity. The **template** is called ***cida*** and its external dependencies are embodied in **windows *p*** and ***g***, expressing the fact that it provides procedures specified in **access interfaces *put*** and ***get***.



The internal structure of **IDA**s created from **template** ***cida*** involves three **components** called ***ip***, ***op*** and ***ex***. Each of these is a **simple IDA** and each is derived from its own **template**. The three **templates** are called ***ip_ida***, ***op_ida*** and ***ex_ida***, respectively. In the more detailed diagram it can be seen from the **window** to **window** connection at the left of the diagram that the interactions associated with the externally visible **window**, ***p***, are provided by a corresponding **window**, named ***pp*** , of the internal **IDA**, ***ip***. In a similar way **window**, ***gg***, of the internal **IDA**, ***op***, is echoed to the outside world at the right of the diagram.

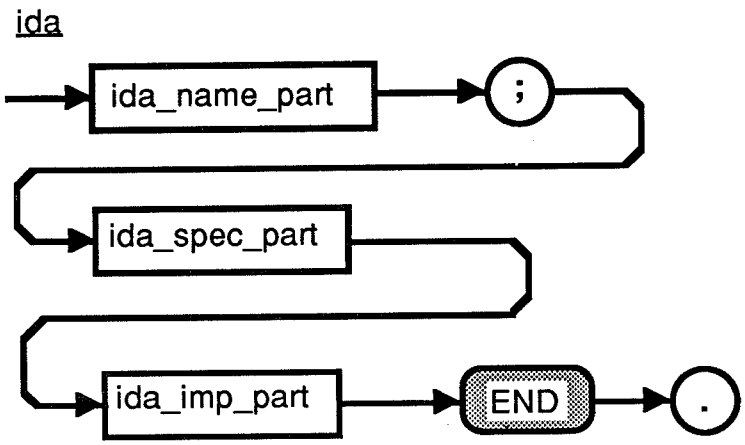
The **IDA**, ***ex***, is a purely private **component** of this **template**. Its single **window**,***ev***, provides facilities, defined by **access interface**, ***vis***, which are required by both of the other **components**. Here, then, is a more complete example of direct **IDA** to **IDA** communication.

Further levels of network decomposition are possible. A thick line boundary to any of the component **IDA**s of a **composite IDA** would indicate this. Such an hierarchical network structure is illustrated in the diagram below in which it is the **component** previously called ***ex*** which has become **composite**. It would be necessary to expand the new **template**, ***cex_ida***, to at least one level down in order to determine the **component** which actually provides the interactions defined in **access interface** ***vis*** in this case.



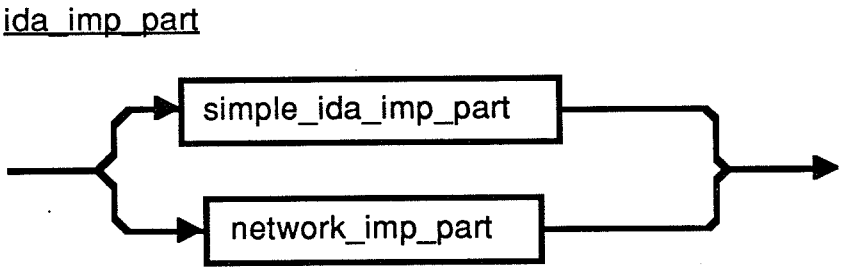
Textual Representation

Where the **module** scheme provides for both **simple** and **composite** forms of **IDA**, the syntax is as follows:



The ***specification part*** is as for the **simple IDA**.

It is in the ***implementation part*** that the distinction between **simple** and **composite** forms occurs.



Like all other **composite** Mascot entities, the **composite** form of an **IDA** contains no explicit coding. It merely defines the nodes and interconnections of a network. The **modules** which represent its component parts, the nodes, are simple **IDA templates** or other **composite IDA templates** or a mixture of both. The interconnections are **paths** represented by **access interfaces**.

The ***name*** and ***specification parts*** are identical to those of the corresponding **simple** form. **Windows** and **ports** are specified in the normal way. The ***implementation part*** takes a similar form to that of a **subsystem**.

Thus the ***implementation part*** begins with a **USES** section which lists the **simple** and **composite IDA templates** from which this **composite IDA** is to be constructed. Referring back to the graphical representation of the **composite IDA** ***cida*** :

```
USES ip_ida, op_ida, ex_ida ;
```

Notice that references to **definitions** are not allowed at this level; there is no code to make use of their contents.

Next comes a list of ***components***. Their syntactic form is as described for **subsystems** earlier in the Handbook except that the allowed **component classes** are **IDA**, **CHANNEL**, **POOL** and **LIBRARY**. Finally, an equivalence list establishes which of the **windows** provided by the internal **components** satisfies each **window** of the enclosing **template**.

To complete this section, the **template** diagram, **cida**, is used as the basis of a summary of the features of the textual representation of a **composite IDA**.

```
IDA cida ; { name part }
        { specification dependencies }
   { CONSTANT ..... ; no template constants }
   PROVIDES p : put ;
            g : get ; { windows }
   { REQUIRES ..... ; no ports }
        { implementation dependencies }
   USES ip_ida, op_ida, ex_ida ; { component templates }
        { components and interconnections }
   IDA ex : ex_ida ;
   IDA ip : ip_ida (ei = ex.ev) ;
   IDA op : op_ida (eo = ex.ev) ;
        { equivalence list }
   p = ip.pp ;
   g = op.gg
END .
```

2.11 COMPOSITE SERVERS

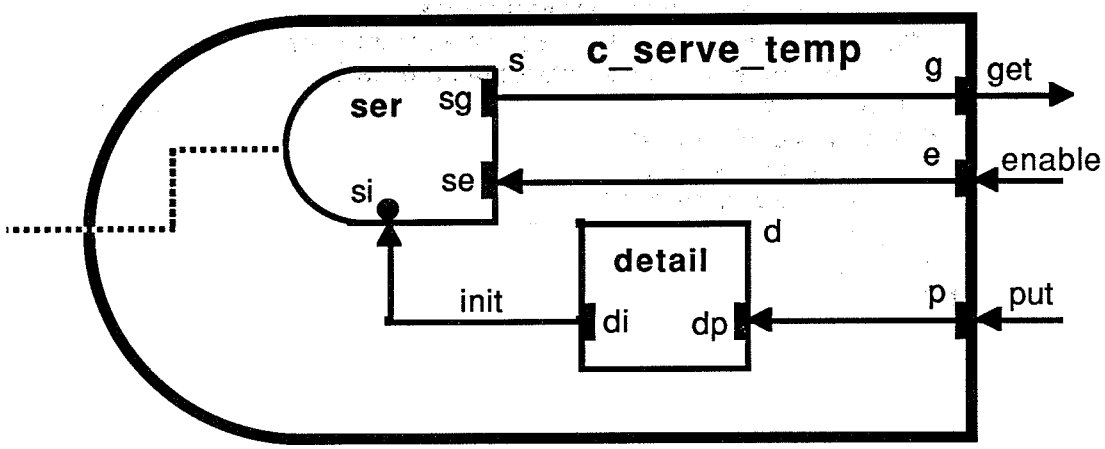
Description

This feature is not part of the mandatory subset of the Mascot definition.

A **composite** form of **server** is provided in Mascot which consists of a **network** containing at least one **simple** or **composite server** and any number of **IDAs**. These **components** communicate with each other by means of **paths**.

Graphical Representation

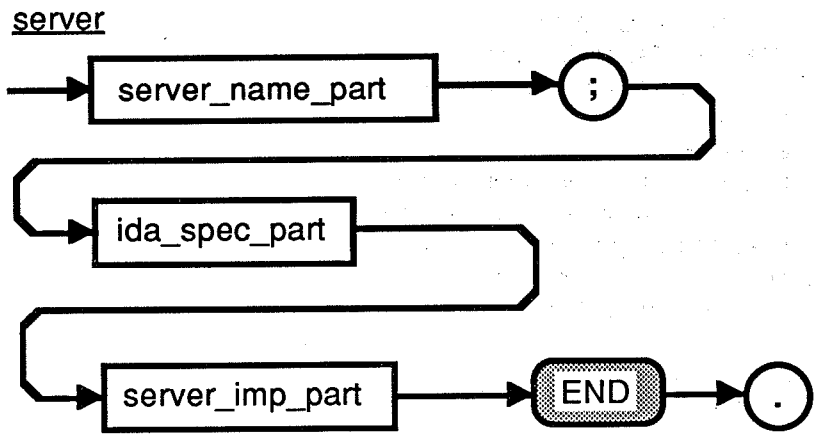
When drawn with a thick line the D symbol is to be interpreted as representing a **composite server**. Its constituent **network** of **servers** and **IDAs**, drawn inside the symbol, completes the representation of the **composite template**.



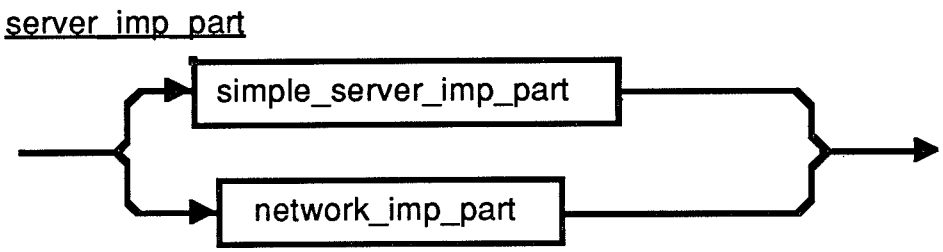
The **template** shown here, ***c_serve_temp***, represents a **composite server** with **components** consisting of one **simple server** and one **simple IDA**. The former, ***s*** is of type ***ser***. Its two **windows**, via **window** to **window** connections, are accessible from outside the **template**. Its single **port** is connected to a **window** of the **IDA component**, ***d*** of type ***detail***, whose other **window** is carried to the outside world.

Textual Representation

Where the **module** scheme provides for both **simple** and **composite** forms of **server** the syntax is as follows:



The ***name*** and ***specification parts*** of a **composite server** are entirely similar to those of the corresponding **simple** form. **Windows** and **ports** may be specified in the normal way. It is in the ***implementation part*** that the two forms are distinguished:



Thus the ***implementation part*** takes the same form as in the **subsystem** and **composite IDA** with the **components** consisting of at least one **server** possibly combined with one or more **IDAs** (which may be **channels** and **pools**) or **libraries**. The **USES** section presents a list of all the **templates** needed to construct the **network**. Then follows the ***component part*** which defines the **network components** and their interconnections. An ***equivalence list*** is used to deal with **window** to **window** connections.

This section ends with an outline example, based on ***c_serve_temp***, of a complete **composite server template**.

```
SERVER ser ;
   PROVIDES sg : get ;
            se : enable ;
   REQUIRES si : init ;
   .
   .
   .
END .

IDA detail ;
   PROVIDES di : init ;
            dp : put ;
   .
   .
END .
```

```
SERVER c_serve_temp ;   { name part }
        { specification dependencies }
   { CONSTANT ..... ; no template constants }
   PROVIDES g : get ;
                e : enable ;
                p : put ;    { window specifications }
   REQUIRES ..... ;        { no port specifications }
        { implementation specifications }
   USES ser, detail ;      { component templates }
        { components and interconnections }
   IDA d : detail ;
   SERVER s : ser (si = d.di) ;
        { equivalence list }
   g = s.sg ;
   e = s.se ;
   p = d.dp
END .
```

2.12 COMPOSITE ACTIVITIES

Description

This feature is not part of the mandatory subset of the Mascot definition.

Activity templates have both the **simple** and **composite** forms. The latter type is not, of course, concerned with **network** decomposition but represents the sequential decomposition of the detailed coding of an **activity**. The principal product of this sequential decomposition is a design element called a **root** which contains the initial entry point of the **composite activity**. The remaining products of decomposition consist of one or more **subroots**.

Communication between the **components** of a **composite activity** is expressed and controlled in a manner analogous to that employed for network interactions. Corresponding to the **path** between the elements of a **network** there is the **link** between the sub-elements of a **composite activity**. A **subelement link**, like a **path**, possesses a type in the form of a **specification**: in this case a **subroot interface** which defines a set of interactions that a **root** or **subroot** is said to need and which is correspondingly given by another **subroot**. The validity of **root** and **subroot** connections is checked in terms of the type of the **link**, the **subroot interface**, exactly as **network** connections are checked in terms of **access interfaces**.

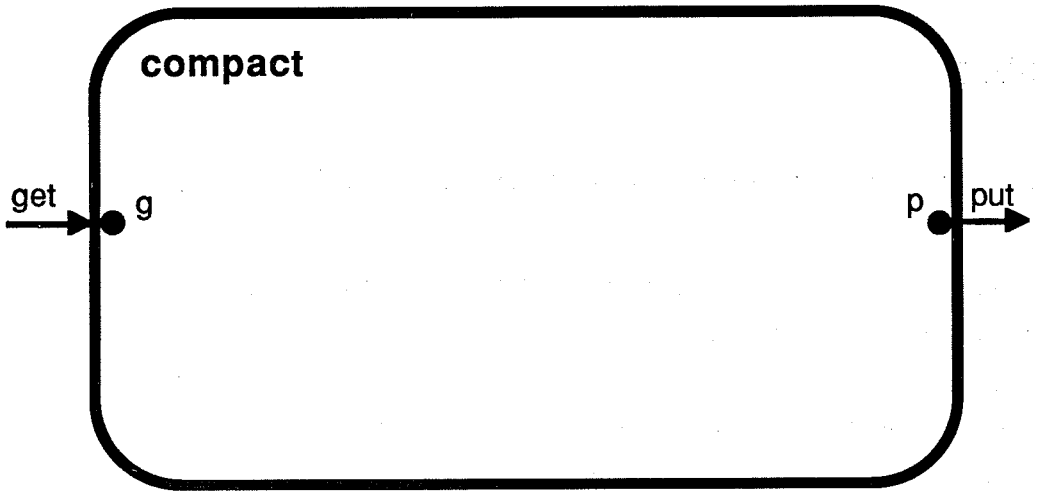
Further levels of sequential decomposition are available through **templates** for **composite subroots**. By this means a **subroot** may by decomposed into a set of internal **subroots** which together perform the same function as a **simple subroot**. That is to say, the ensemble is able to give the interactions specified in exactly one **subroot interface**.

External network connections may be made from any **component** of a **composite activity**. Thus **ports** may be specified in a **root** or in any **subroot** to correspond with those specified in the **template** which describes the **composite activity** at its outermost level. The nature of this correspondence is discussed in more detail in connection with the graphical and textual forms of representation.

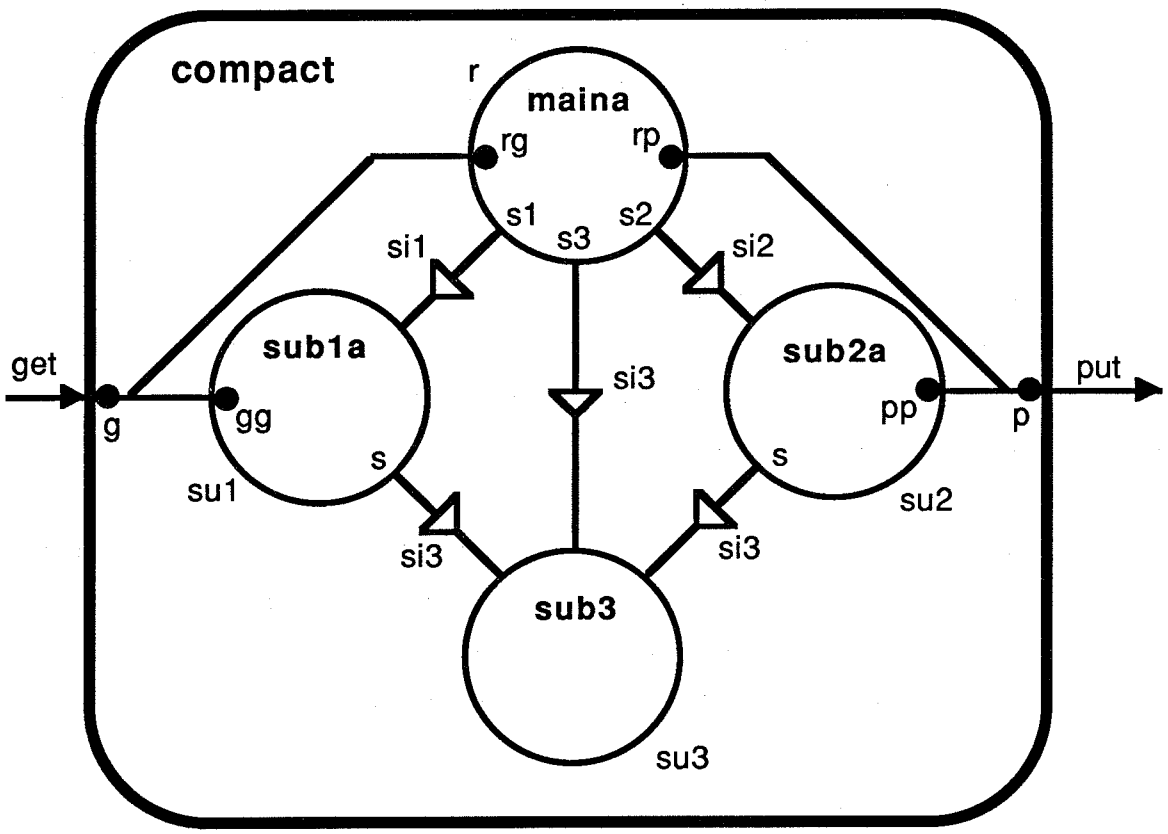
Graphical Representation

The graphical conventions for **roots** and **subroots,** in **composite activities**, are similar to those for **simple activities**. They are normally represented by circular symbols to which **port** connections may be made but they also possess sub-element **links** to illustrate sequential decomposition. The latter connections are shown as thin lines broken by hollow arrow heads which indicate the direction of procedure invocation. The following diagram illustrates a **template** for a **composite activity**. Its

composite nature is indicated by the use of a thick line for its boundary.



The rounded corners proclaim that it defines an active design element. The **template** is called ***compact*** and its dependencies are embodied in **ports** ***g*** and ***p*** expressing requirements specified by **access interfaces** ***get*** and ***put***.

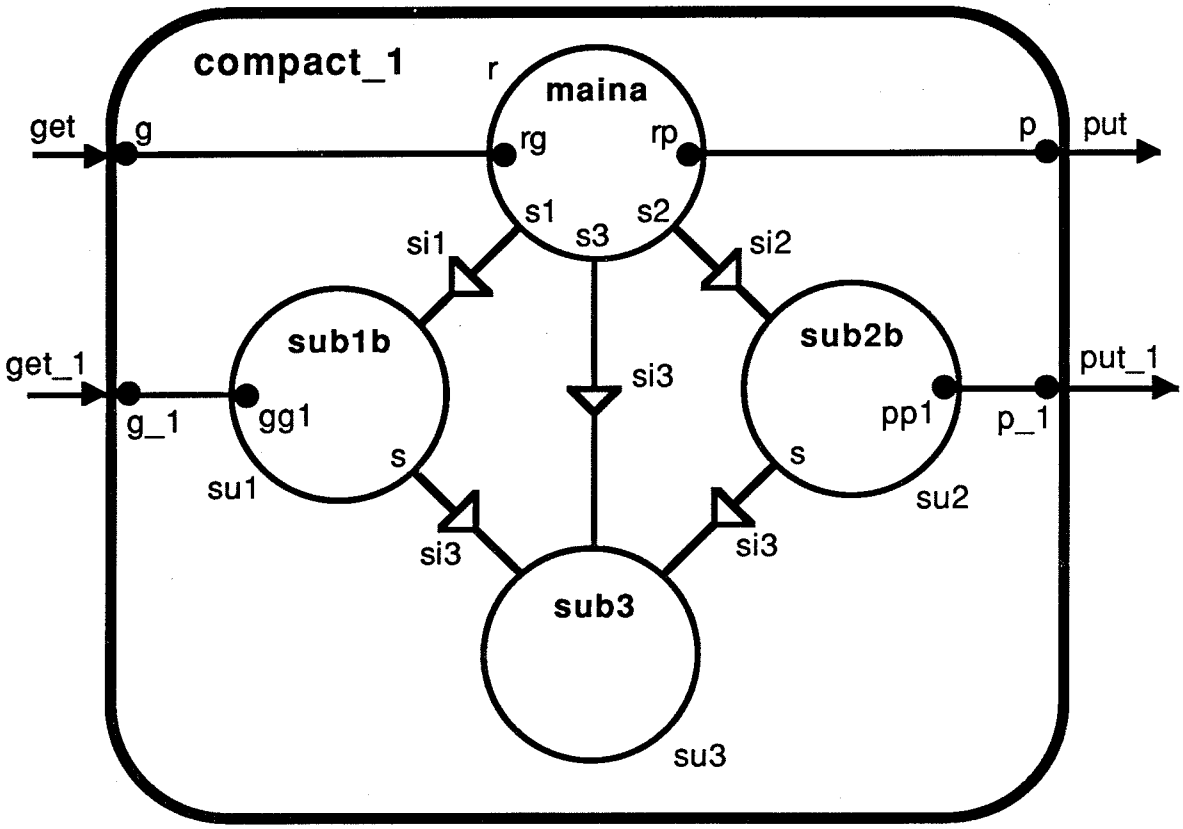


The internal structure of **activities** created from this **template** involves four **components**, a **root** called ***r*** and three **subroots** called ***su1***, ***su2*** and ***su3***. The **templates** which define these four **components** are called ***maina*** , ***sub1a***, ***sub2a*** and ***sub3***, respectively. Three **subroot interfaces**, ***si1***, ***si2*** and ***si3*** specify the interactions which the three **subroots** make available via sub-element **links**. ***r*** makes use of all three sets of interactions while ***su1*** and ***su2*** utilise only the set

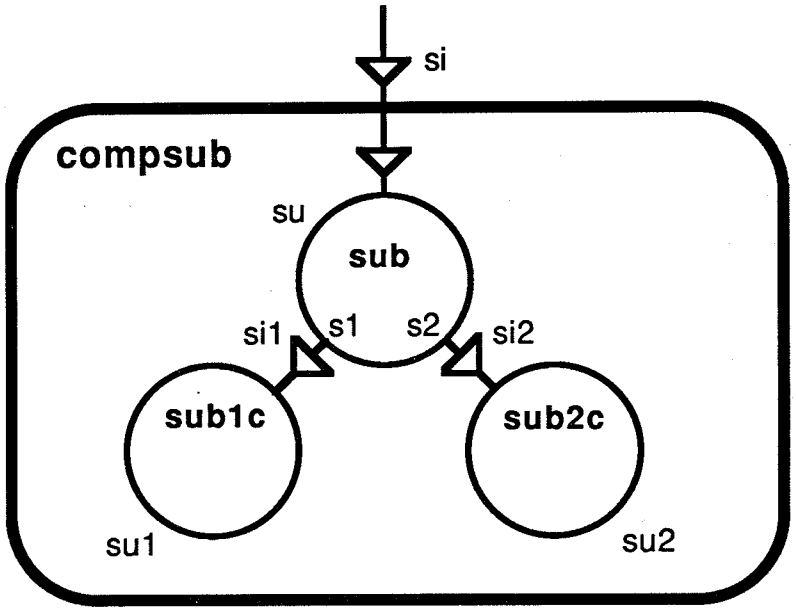
defined in ***si3***.

The manner in which these internal **components** relate to the **template**'s external dependencies is shown by the **port** to **port** connections which continue through the outer boundary. From these it can be seen that both ***r*** and ***su1*** utilise the operations defined in **access interface** ***get*** and that ***r*** and ***su2*** use those in **access interface** ***put***.

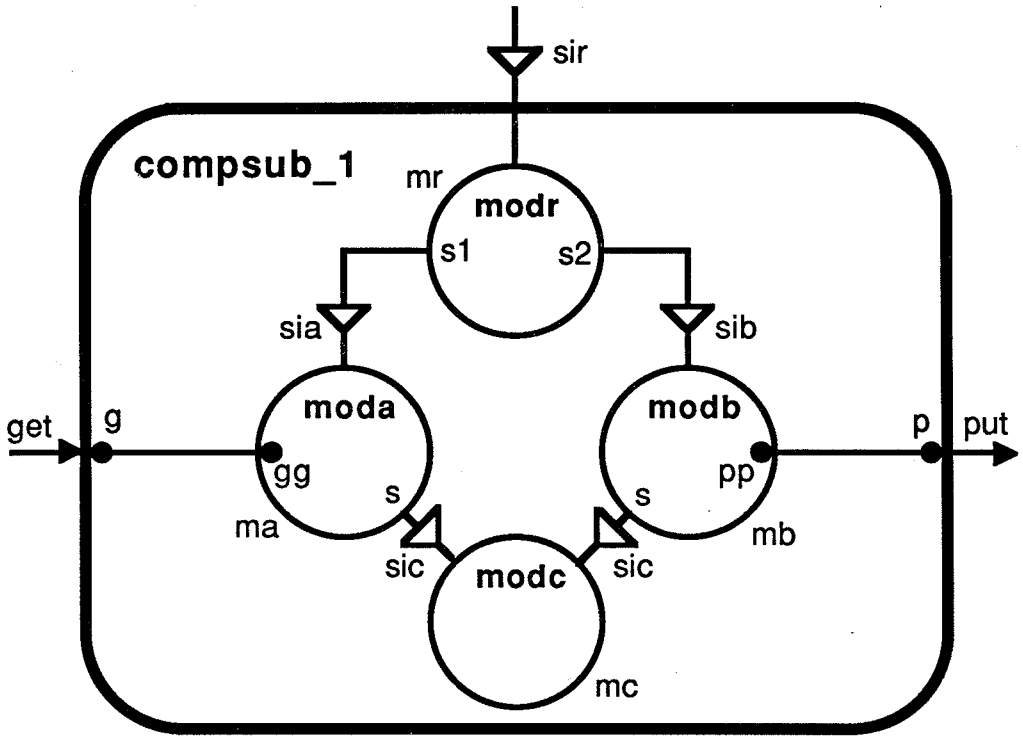
Since **roots** and **subroots** contain sections of coding which are all conceptually part of the same **activity**, **ports** may be established anywhere in the internal structure. This is illustrated in the **template** ***compact_1***, below.



Further levels of the sequential decomposition of **activities**, in the form of **composite subroots**, possess a similar graphical representation. However, a **composite subroot** will show a **link** which penetrates its boundary and terminates on a **simple** internal **root component** derived from an appropriate **subroot template**.



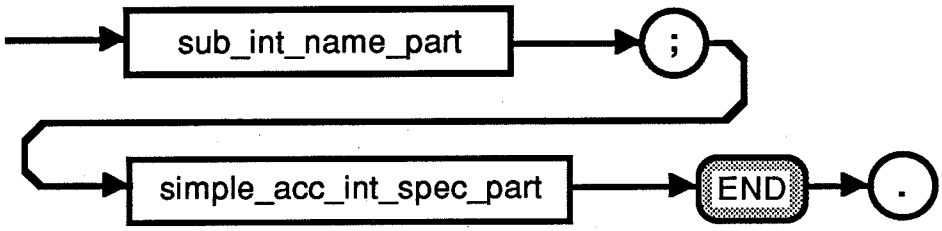
Subroots established at these deeper levels of nesting retain all the usual properties inherited from **activities**. Thus, on the diagram below, two **components** of the **composite subroot** possess **ports**.



Textual Representation

Before describing the textual form of a **template** for a **composite activity** it is necessary first to discuss the **templates** for its component parts and their interconnections, namely **roots, subroots** and **subroot interfaces**. The latter has a structure similar to that for an **access interface**.

subroot_interface



The ***name part*** employs the appropriate alternative language word to establish the **class** of the **module**.

sub_int_name_part

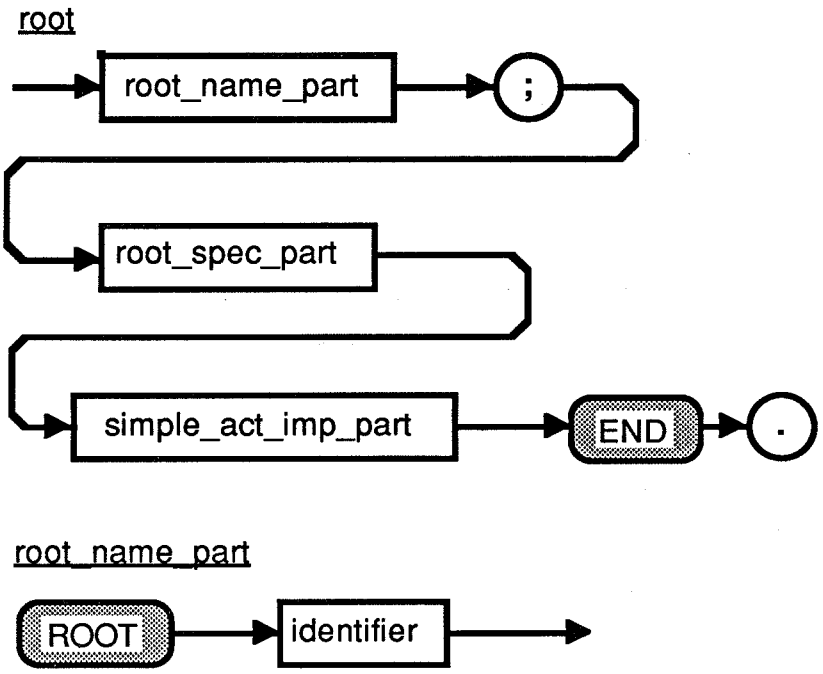
while the remainder of the **module** is exactly as defined for the **access interface** since the ***specification part*** is common to these two types of Mascot **interface**.

The following examples illustrate some of the possible forms :

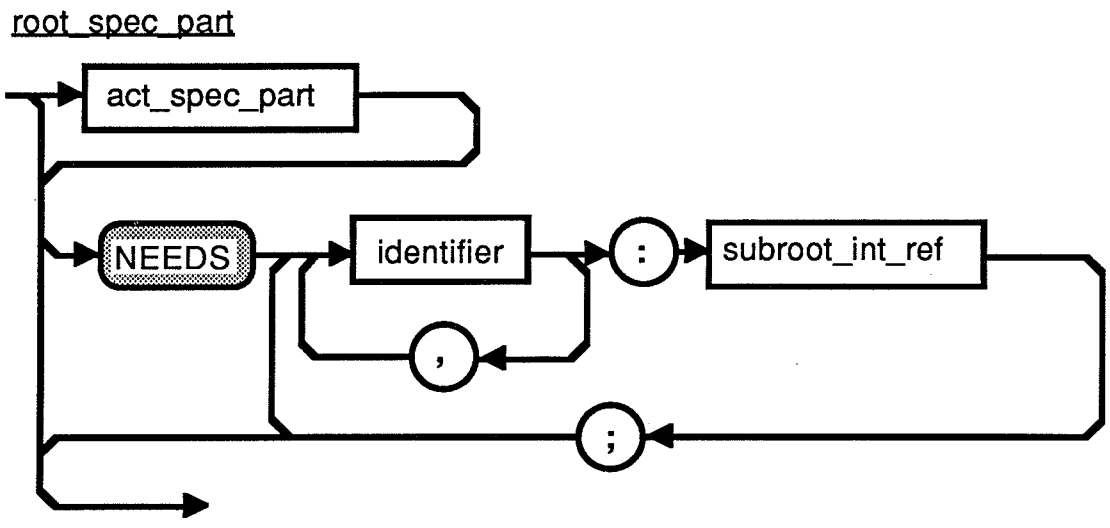
```
{ subroot interface with procedure specifications }
        SUBROOT INTERFACE locprocs ;
           FUNCTION factorial( i : integer ) : integer ;
           FUNCTION modulo( i, j : integer ) : integer ;
        END .

{ subroot interface with definition dependency }
        DEFINITION conventions ;
           TYPE
              vector = RECORD
                          x_coord : real ;
                          y_coord : real ;
                          z_coord : real
                       END;
              direction_cosines = RECORD
                                     cos_alpha : real ;
                                     cos_beta : real ;
                                     cos_gamma : real
                                  END;
        END .

        DEFINITION diagram ;
           TYPE
              diagram = RECORD

                        END;
        END .
```

```
SUBROOT INTERFACE geometry ;
   WITH conventions, diagram ;
   PROCEDURE translate( d : diagram; v : vector ) ;
   PROCEDURE rotate( d : diagram; dc : direction_cosines ) ;
END .
```

A **template** for a **root** naturally bears a close resemblance to a **template** for a **simple activity**. **Ports** and **template constants** may appear in the ***specification part*** ; **definition** and **library** dependencies may be expressed in the ***implementation part*** whose block section also contains the initial entry point of the **composite activity** of which it is a **component.**



The ***specification part*** of a **root module** differs from that representing a **simple activity** only in its ability to express dependencies which are satisfied through **subelement links**. The following diagram shows how this feature is incorporated.



There follows a possible outline for the **root module** in the **composite activity template** ***compact*** shown in graphical form earlier.

```
ROOT maina ; { name part }
        { specification dependencies }
   { CONSTANT ..... ; no template constants }
   REQUIRES rg : get;
            rp : put ;          { ports }
   NEEDS s1 : si1;
         s2 : si2 ;
         s3 : si3 ;          { outward subroot links }
        { implementation dependencies }
   WITH ..... ;              { global definitions }
   { LIBRARY ..... ; no library dependencies }
      { local declarations }
   BEGIN
      { statement sequence }
   END
END .
```

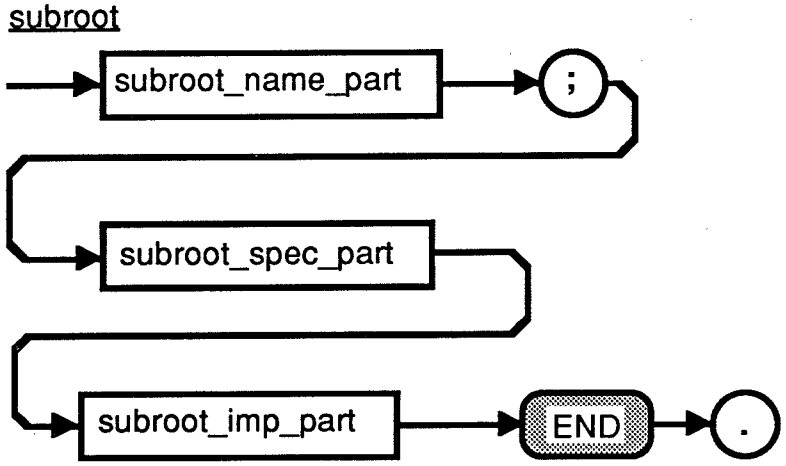
The syntax of a **subroot module** differs from that of a **root** in ways which reflect the following different properties :

(a) A **subroot** may appear at either end of a **subelement link**. That is, it may implement facilities for use by a **root** or by other **subroots** as well as using such facilities.

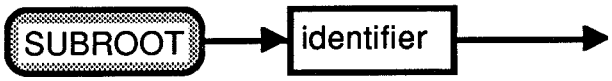
(b) A **subroot** may be **composite**.

(c) Unlike a **root**, a **subroot** cannot be entered for execution directly. The executable code that it contains is all encapsulated in procedures which are called through a **subroot interface**. It therefore has no outer block statement sequence.

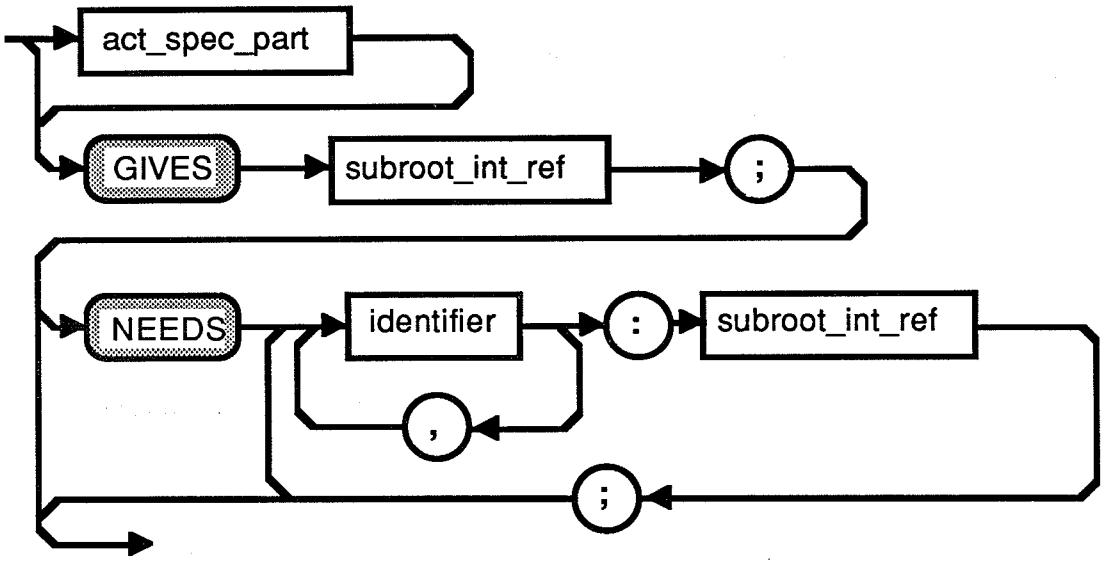
Difference (a) implies an addition to the ***specification part***, compared with a **root module**, and (b) and (c) involve variations in the ***implementation part***. As elsewhere, the outline syntax is presented first with an expansion of the ***name*** and ***specification parts***.



subroot_name_part



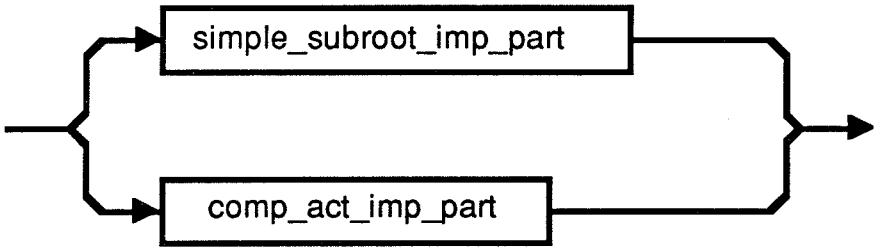
subroot_spec_part



The diagram describing the ***specification part*** shows that, after optionally specifying **ports** and **template constants**, a **GIVES** section identifies the **subroot interface** which is implemented by this **template**. It should be noted that there is precisely one such **interface**. On the other hand, like a **root**, a **subroot** may utilise (keyword **NEEDS**) the services of any number of other **subroots** through **subelement links**.

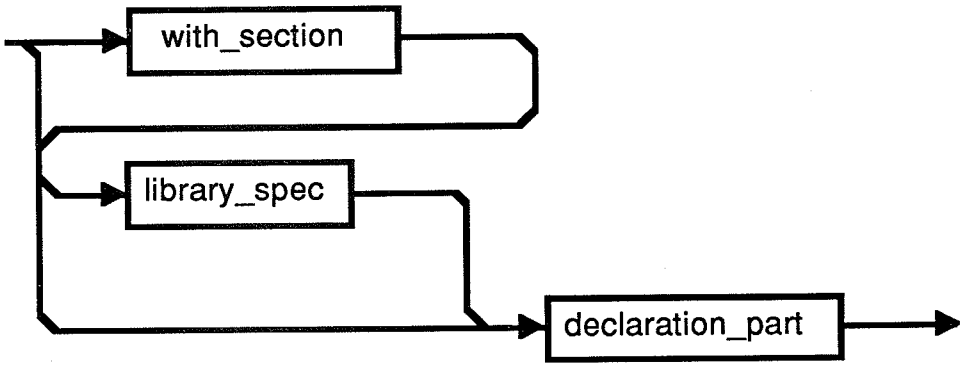
Turning now to the ***implementation part*** ; a distinction has to be made between the **simple** and **composite** forms :

subroot_imp_part



For a **simple subroot** the ***implementation part*** is similar to that for a **simple activity** except for the omission of the statement sequence.

simple_subroot_imp_part



Thus, global definitions and **library** sub-threads may be used exactly as in **simple activities** and in **roots**. In the ***declaration part,*** constants, types and variables, private to the **subroot**, may be defined. These are followed by definitions of the procedures which are specified in the **subroot interface** mentioned in the **GIVES** section, together with any purely private procedures which the **interface** procedures use.

It is now possible to provide, in outline, the text needed to represent **templates** for each of the **components** of the **composite activity** illustrated earlier as ***compact***. But first the **interfaces** must be specified. Two **access interfaces** are required for external **network** communication and three **subroot interfaces** for internal communication between the **root** and **subroot components**.

```
DEFINITION network_data_def ;
   TYPE
      network_data = ..... ;
END .

ACCESS INTERFACE put ;
   WITH network_data_def ;
   PROCEDURE write( item : network_data ) ;
END .

ACCESS INTERFACE get ;
   WITH network_data_def ;
   FUNCTION read : network_data ;
END .

DEFINITION subroot_data_def ;
   TYPE
      subroot_data = ..... ;
END .

SUBROOT INTERFACE si1 ;
   WITH network_data_def, subroot_data_def ;
   FUNCTION process_1( item : network_data ) : subroot_data ;
END .
```

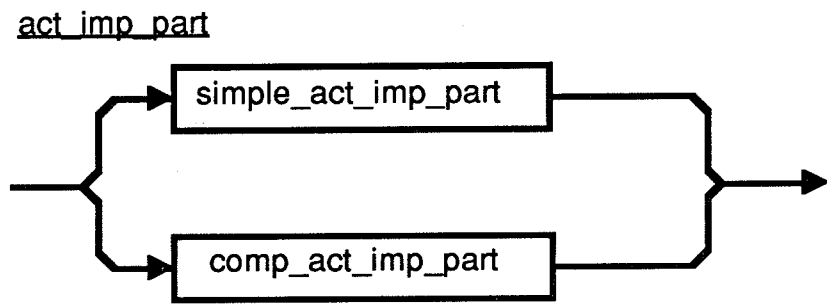
```
SUBROOT INTERFACE si2 ;
   WITH network_data_def, subroot_data_def ;
   FUNCTION process_2( item : subroot_data ) : network_data ;
END .

SUBROOT INTERFACE si3 ;
   PROCEDURE calculate( in : integer ; VAR out : integer ) ;
END .
```

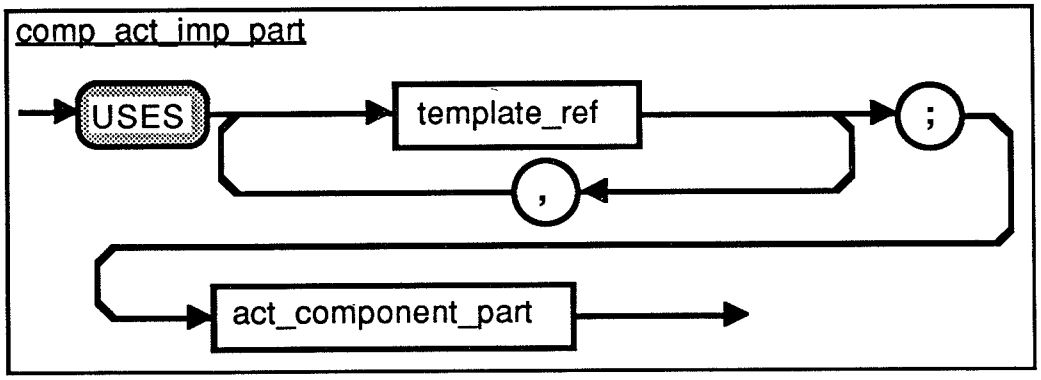
Templates for the **root** and the three **subroots** can now be outlined:

```
ROOT maina ;
        { specification dependencies }
   { CONSTANT ..... ;      no template constants }
   REQUIRES rg : get ;
            rp : put ;                    { ports }
   NEEDS s1 : si1 ;
         s2 : si2 ;
         s3 : si3 ;    { outward subroot links }
        { implementation dependencies }
   WITH ..... ;                           { global definitions }
   { LIBRARY ..... ;      no library dependencies }
        { local declarations }
        { root coding with initial entry point  }
END .

SUBROOT sub1a ;
        { specification dependencies }
   { CONSTANT ..... ;      no template constants }
   REQUIRES gg : get ;      { ports }
   GIVES si1 ;                            { inward subroot link }
   NEEDS s : si3 ;                        { outward subroot link }
        { implementation dependencies }
   WITH ..... ;                           { global definitions }
   { LIBRARY ..... ;      no library ddependencies }
        { interface procedures }
   FUNCTION process_1( i : network_data ) : subroot_data ;
      .
      .
END .

SUBROOT sub2a ;
        { specification dependencies }
   { CONSTANT ..... ;      no template constants }
   REQUIRES pp : put ;      { ports }
   GIVES si2 ;                            { inward subroot link }
   NEEDS s : si3 ;                        { outward subroot link }
        { implementation dependencies }
   WITH ..... ;                           { global definitions }
   { LIBRARY ..... ;      no library dependencies }
        { interface procedures }
   FUNCTION process_2( i : subroot_data ) : network_data ;
      .
      .
END .
```

```
SUBROOT sub3 ;
        { specification dependencies }
   { CONSTANT ..... ;     no template constants }
   { REQUIRES ..... ;   no ports used }
   GIVES si3 ;                                  { inward subroot link }
   { NEEDS ..... ;     no outward subroot links }
        { implementation dependencies }
   WITH ..... ;                                 { global definitions }
   { LIBRARY ..... ;     no library dependencies }
        { interface procedures }
   PROCEDURE calculate( in : integer ; VAR out : integer );
   .
   .
END .
```

To see how to express the **template** for ***compact*** itself, it is necessary to return to the discussion of **activities** and, in particular, to the **composite** form which has yet to be described. The ***name*** and ***specification parts*** are entirely similar to those of the corresponding **simple template**. Thus, **ports** may be specified in the normal way. The ***implementation part***, however, as in the case of a **root**, takes alternative forms for **simple** and **composite activities**.



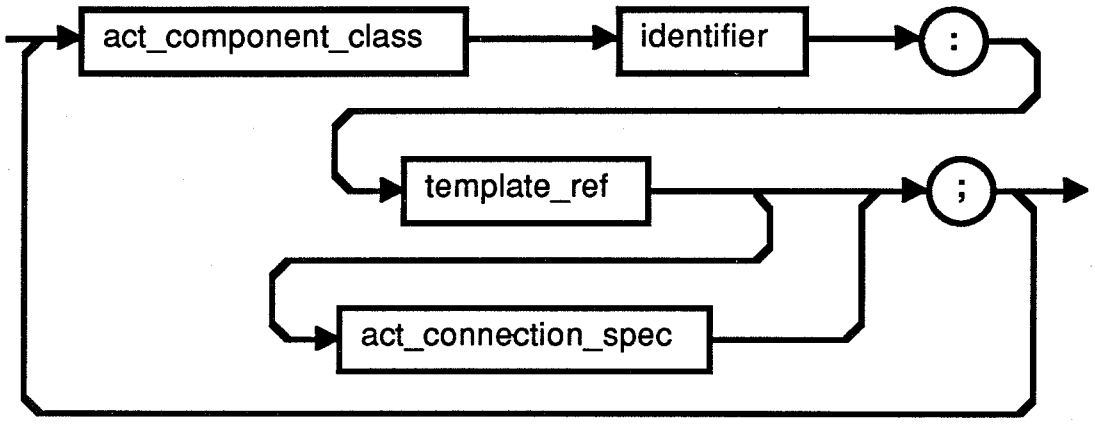


It begins with a **USES** section which lists the **root template** and the highest level **subroot templates** from which the **activity** is to be constructed. Referring back to the graphical representation of the composite **activity** ***compact*** :

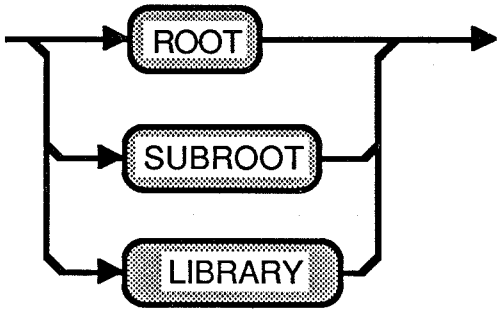
```
USES maina, sub1a, sub2a, sub3 ;
```

This indicates which **templates** are needed to create the component parts of the **activity**. Notice that references to **definitions** are not allowed at this level. The ***implementation part*** contains no local declarations or program statements. This is a **composite module** and its purpose is to specify the internal structure of the **activity** in terms of **root** and **subroot components**, created from the **templates** mentioned in the **USES** section, and the **subroot interfaces** through which they communicate.

act_component_part

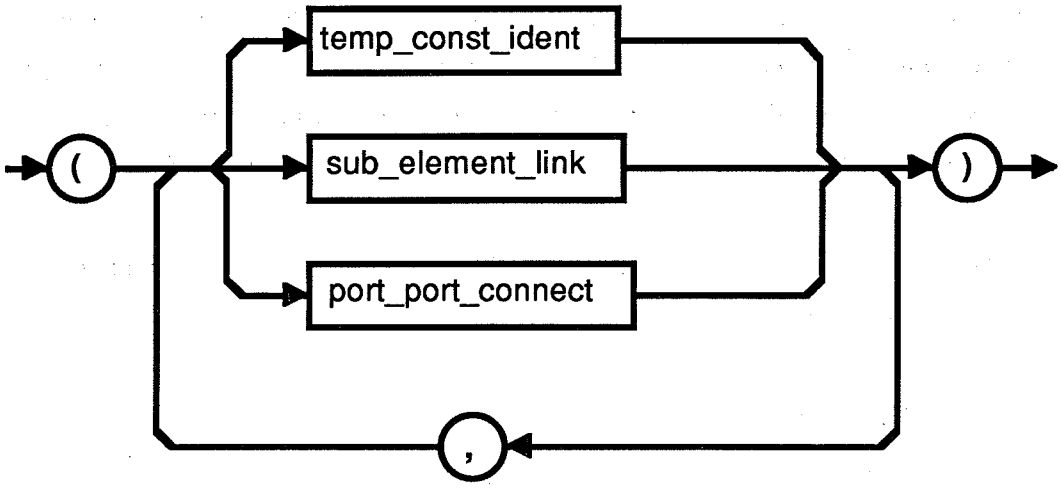


act_component_class



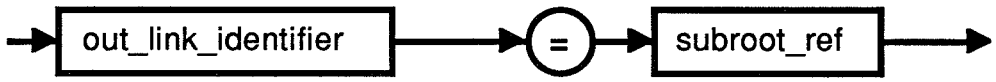
Every **composite activity** specifies a single **root component** together with any number of **subroots** and any **libraries** which are to be made available to the **components**. The specification of each **component** defines any connections it possesses to **ports** of the **activity** and any **links** to **subroots**.

act_connection_spec



The general form of the connection specification is very similar to that already encountered for a **subsystem**. The syntax of **port** to **port** connections and **template constant** identities have already been described in earlier sections of the Definition. The **subelement link** syntax is given below:

sub_element_link



Each outward **link** of the **component** being defined is equated with the **subroot component** which is to give the required interactions.

The **template** for the **composite activity** ***compact*** may now be written:

```
ACTIVITY compact ;
        { specification  dependencies }
   { CONSTANT ..... ;              no template constants }
   REQUIRES  g : get ;
             p : put ;        { ports }
        { implementation  dependencies }
   USES maina, sub1a, sub2a, sub3 ; { component templates }
        { components  and  interconnections }
   SUBROOT su1 : sub1a (  gg = g,
                          s = su3 ) ;
   SUBROOT su2 : sub2a (  pp = p,
                          s = su3 ) ;
   SUBROOT su3 : sub3 ;
   ROOT r : maina (    rg = g,
                       rp = p,
                       s1 = su1,
                       s2 = su2,
                       s3 = su3 ) ;
END .
```

One topic remains to be covered to complete the syntactic description of **activities**. This is the **composite** form of the **subroot**. Like the **composite activity**, it differs from its **simple** form only in the ***implementation part***. Furthermore, this differs from the corresponding ***part*** of a **composite activity** only in that it cannot contain a **component** derived from a **root template**. Instead, the **root component** is derived from a **subroot template** which gives the appropriate **interface**.

The **template** diagram, ***compsub***, at the end of the description of graphical representation will now be used to illustrate the application of these syntax rules.

```
SUBROOT compsub ;
        { specification dependencies }
   { CONSTANT ..... ;  no template constants }
   { REQUIRES ..... ;  no ports }
   GIVES si ;                  { inward subroot link }
   { NEEDS ..... ;   no outward subroot links }
        { implementation dependencies }
   USES sub, sub1c, sub2c ;     { component templates }
        { components and interconnections }
   SUBROOT su1 : sub1c ;
   SUBROOT su2 : sub2c ;
   ROOT su : sub ( s1 = su1,
                   s2 = su2 );
END .
```

2.13 ARRAYS OF PORTS AND WINDOWS

Description

This feature is not part of the mandatory subset of the Mascot definition.

Templates for both the **simple** and **composite** forms of **IDAs**, **activities** and **servers** together with **subsystems**, **roots** and both **simple** and **composite subroots** may specify **ports** by means of a **REQUIRES** section in their ***specification parts*** . The identifiers so introduced represent **ports** which may be referred to in the code sections of the **simple templates**. In some cases it may be required to transmit data via each member in turn of a group of **ports** from a program loop. It is therefore valuable to be able to specify such a group as an array. Since all or some of the **paths** associated with an array of **ports** may pass through the boundary of an enclosing **composite** design element it is also useful to be able to use arrays at that level.

Of the above list of **template** types, **simple** and **composite IDAs** and **servers** together with **subsystems** may also specify **windows** by means of a **PROVIDES** section. These identifiers are used in the ***connection specifications*** of **simple templates** and the ***equivalence lists*** of both forms. Here again it is useful to have arrays in order to express logical groupings of **windows**.

The complete definition of Mascot caters for both these requirements.

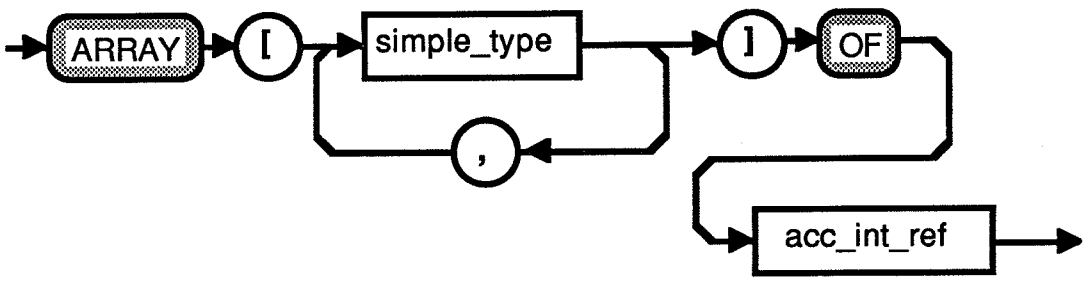
Graphical Representation

An array of **ports** or **windows** may be representeds by a single symbol or, alternatively, each element may be shown individually. Where the elements are shown individually the array index should also appear.

Textual Representation

The extension of the design language syntax to accommodate arrays of **ports** and **windows** in **REQUIRES** and **PROVIDES** lists (see Section 2.3) is shown below:

acc_int_array_descrip



Thus the **REQUIRES** list of an **activity**, say, might be:

```
REQUIRES p_set : ARRAY[1 .. 5] OF put ;
```

where ***put*** is an **access interface**. This would facilitate coding such as:

```
FOR line := 1 TO 5 DO
BEGIN
   .
   p_set[line] . send(item[line]) ;
   .
END
```

where ***send*** is a procedure specified in the **access interface**.

An array of **windows** in an **IDA**, say, might be specified as:

```
PROVIDES w_set : ARRAY[1 .. 3] OF get ;
```

and utilised in an ***equivalence list*** as:

```
w_set[1] . fetch = fetch1 ;
w_set[2] . fetch = fetch2 ;
w_set[3] . fetch = fetch 3
```

to designate three separate **ACCESS** procedures for use in connection with three logically related **paths**.

2.14 COMPOSITE PATHS, PORTS AND WINDOWS

Description

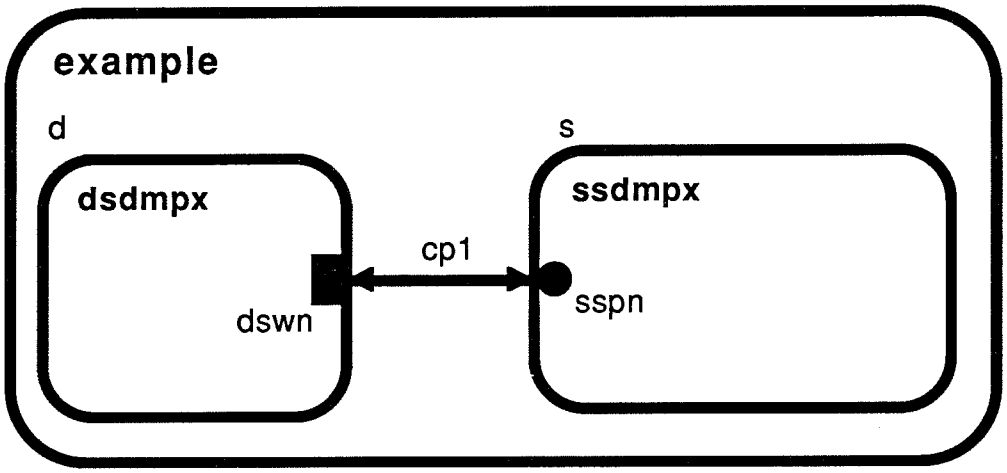
This feature is not part of the mandatory subset of the Mascot definition.

Mascot allows a system design to be expressed in an hierarchical form such that components at the top level are decomposed into the more detailed components at lower levels of the hierarchy. It follows that the granularity of the interactions on a **path** in network diagrams, at different levels in the hierarchy, may vary. The definition presented so far has only provided for one level of granularity to be used for **paths**; that is the **access interface**'s procedures. The **composite path** is provided, in the full Mascot definition, to allow different levels of granularity to be represented.

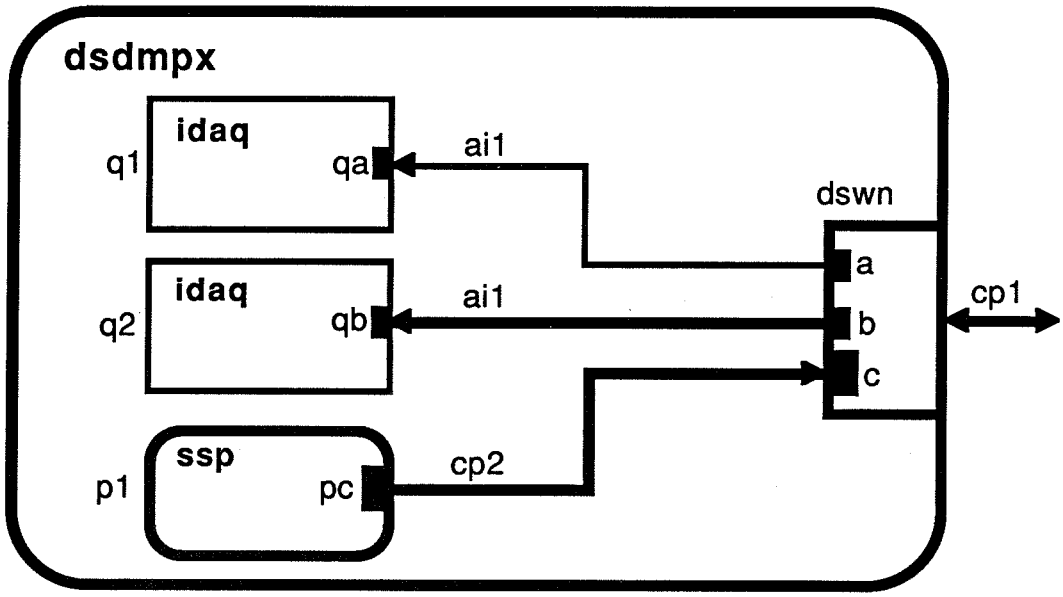
The basic concept is that a **composite path** represents a 'trunk' route between two **subsystems** of a **network**. The **module** which defines the type of a **composite path** is an **access interface** which itself comprises a set of lower level **access interfaces**. A **composite path** is treated exactly like a **simple path** until it needs to be split into its component parts, at which points, a special form of 'adaptor' is used: the **composite port** or the **composite window**.

Graphical Representation

A **composite path** is represented on a Mascot network diagram by a thick line connecting a **port** of one **component** with a **window** of another.

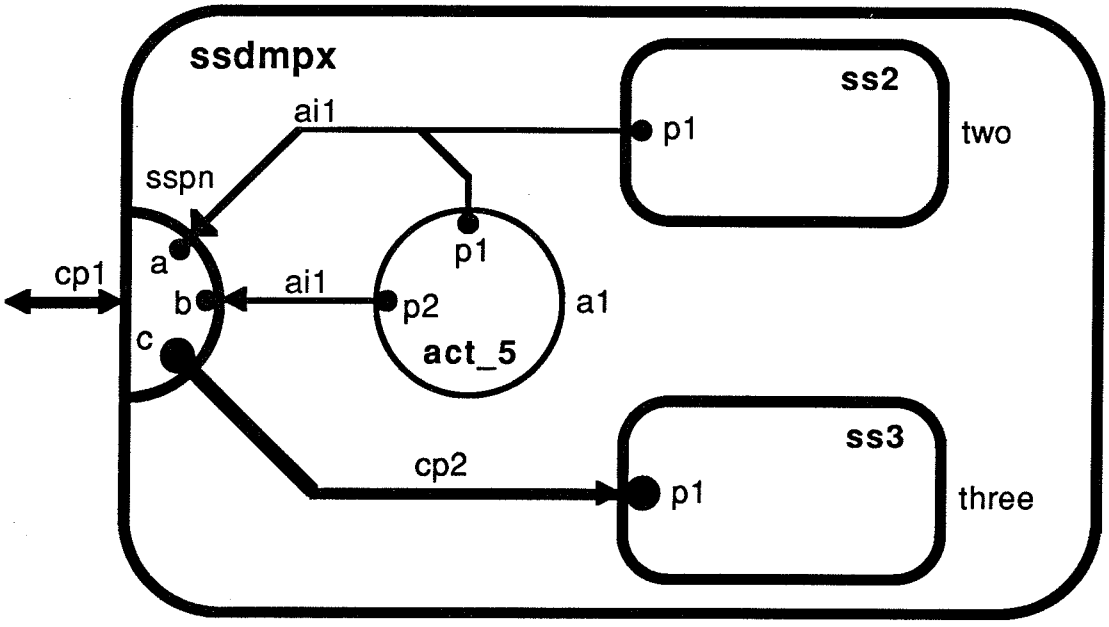


This is illustrated in the diagram above in which the two **components** are a **subsystem** ***s*** and a **subsystem** ***d*** . The name of the **composite path** which connects them is ***cp1***. The **template** for the **subsystem** ***dsdmpx***, where the decomposition takes place, shows how the **composite window** ***dswn*** is represented.



Like a **simple window**, the **composite window** is drawn as a rectangular shape. However it is hollow and, like all **composite** entities, it has a thick outline. It can only occur on the boundary of a **subsystem template**. On the outside of the **template** to which it belongs there is an external connection labelled with the name ***cp1*** of the **composite access interface** which describes, indirectly, the interactions it provides. Inside the **composite window** symbol the individual component **windows** are shown and labelled in the normal way. In this case, two of them, ***a*** and ***b*** , are **simple** and are connected via **simple paths**, both of type ***ai1***, to **simple IDA components** ***q1*** and ***q2*** . The third is itself **composite** and is connected via the **composite path** ***cp2*** to a **subsystem component** ***p1*** . It can be seen from this diagram how a **composite window** acts as an adaptor to 'demultiplex' a **composite path**.

The **template** for **subsystem** ***ss*** shows how the **composite port** ***sspn*** is drawn.

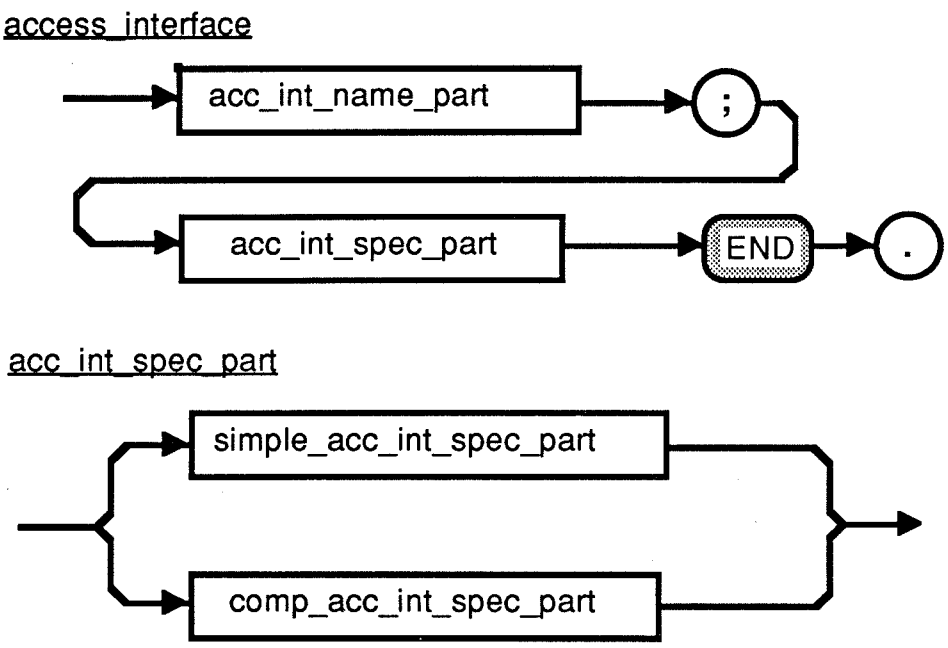


The **composite port** ***sspn*** is shown as a hollow semi-circle, with a thick boundary, inside which are the 'demultiplexed' **ports** ***a***, ***b***, and ***c*** which it makes visible. Two of these are **simple** and are connected via **paths** of type ***al1*** to corresponding **ports** of the **subsystem's components**. ***c*** is itself **composite** and is connected via the **composite path** ***cp2*** to a **port** of the **component** ***three*** . The external connection is labelled to show that the interactions which it requires are defined, indirectly, by the **composite access interface** ***cp1*** . Since a **composite port** connected to a **composite window** must be defined by the same **access interface**, not only the types but also the names of the constituent **ports** and **windows** must match in order that constituent **ports** and **windows** of the same type may be distinguished from each other.

The symbols used to represent **composite ports** and **windows** may be drawn larger than the normal size so as to be more in proportion to the thickness of the **path**. This is illustrated by the **window*c*** and the **ports** ***p1*** and ***c*** in the above diagrams.

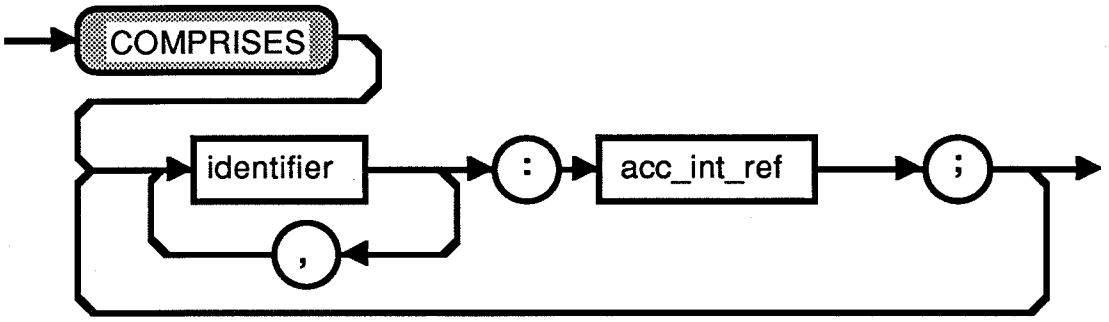
Textual Representation

The **module** which defines the operations, required by a **port**, provided by a **window**, and hence defines the interactions along a **path** is the **access interface**. This is a **specification** which if **composite** defines, indirectly, the possible interactions by specifying the set of **access interfaces**, **simple** or **composite**, of which it is comprised. The complete syntactic structure for an **access interface** shows how the **composite** form is catered for:



The ***specification part*** of the **composite** form has the following structure:

comp_acc_int_spec_part



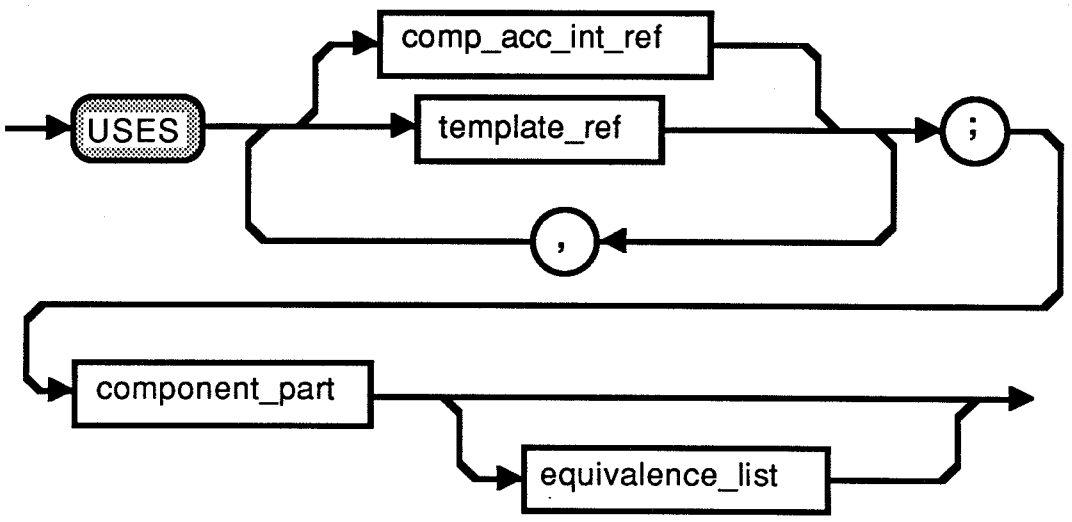
Taking the **interface *cp1*** from the above graphical examples:

```
ACCESS INTERFACE cp1;
   COMPRISES a, b : ai1;
             c : cp2;
END.
```

ai1 it will be recalled is a **simple interface** while ***cp2*** is **composite**.

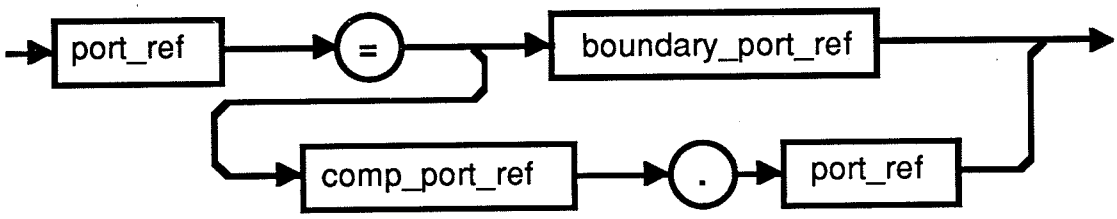
A **port** or **window** to handle a **composite path** arises in the normal way as a reference in a list following the keywords **REQUIRES** or **PROVIDES**. However in order to indicate that a **composite path** is to be decomposed within a **composite template**, the **composite access interface** must be referred to in the **USES** list of the **template**. The syntax is:

network_imp_part



The manner of expressing the connections which fan out from **composite ports** is revealed by the complete syntax diagram for a **port** to **port** connection as it appears in a connection specification:

port_port_connect

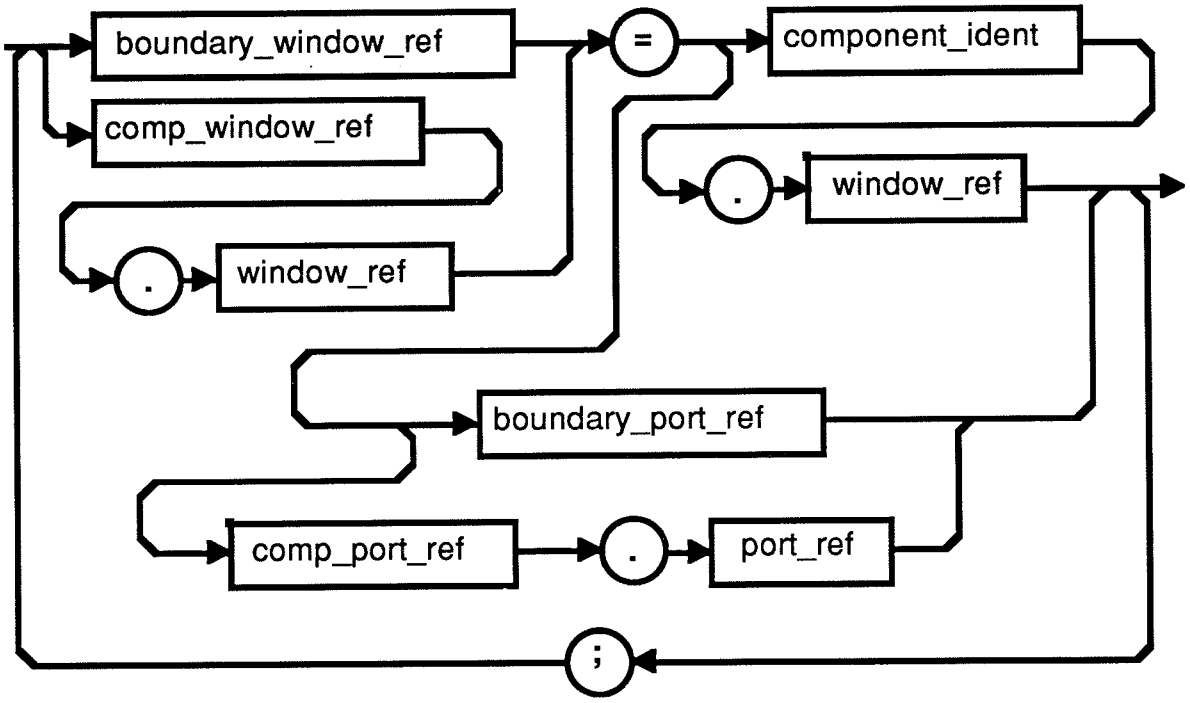


Returning to the earlier graphical examples, the following is the textual form of the **subsystem *ssdmpx***

```
SUBSYSTEM ssdmpx;
   REQUIRES sspn : cp1;

   USES cp1, ss2, ss3, act;
   SUBSYSTEM two : ss2 (p1 = sspn.a);
   SUBSYSTEM three : ss3 (p1 = sspn.c);
   ACTIVITY a1 : act_5 (   p1 = sspn.a,
                           p2 = sspn.b);
END.
```

At the other end of a **composite path**, connections fan out from a **composite window**. These, being **window** to **window** connections, are defined in an equivalence list for which the complete syntax diagram is:

equivalence_list



Thus the **template** for the **subsystem *dsdmpx*** in the graphical examples is expressed as:

```
SUBSYSTEM dsdmpx;
   PROVIDES dswn : cp1;

   USES cp1, idap, idaq;
   IDA q1 : idaq;
   IDA q2 : idaq;
   SUBSYSTEM p1 : ssp;

   dswn.a = q1.qa;
   dswn.b = q2.qb;
   dswn.c = p1.pc;
END.
```

and for completion here is the **template** for the **system** which connects the **subsystem** and the **composite IDA** together:

```
SYSTEM example;
   USES dsdmpx, ssdmpx;

   SUBSYSTEM d : dsdmpx;
   SUBSYSTEM s : ssdmpx (sspn = d.dswn);
END.
```

2.15 DIRECT DATA VISIBILITY

Description

This feature is not part of the mandatory subset of the Mascot definition.

One of the essential objectives of Mascot is to control access to shared data from concurrently executing **activities**. The responsibility for this control, and hence for the integrity of the shared data, is normally exercised solely by the **IDA**. In special circumstances, however, it may be necessary to bypass the protection normally afforded by the **access procedure** mechanism and to allow **activities** to manipulate **IDA** data directly. There are two classes of problem which may be solved by this means:

1. The first problem concerns the location of code and data on multi-processor configurations. It may be necessary to locate shared data, that is **IDA** data areas, and the code which operates on them in separate areas of memory. This problem may also arise in single processor configurations which possess non-homogeneous memory. A possible solution is to use a **composite IDA** and to provide direct data visibility in the **interfaces** between its **components**. Instances of the **component IDAs** can be placed in the appropriate memory locations during the **building** of the application software.

2. The second problem is concerned with efficiency of access to data. There are some occasions when the overhead associated with the call of an **access procedure** far outweighs the processing time and memory space required by the access mechanism code itself. If this applies in a **path** along which data transfers occur at a high frequency, the overall efficiency may become unacceptably low. In such cases direct data visibility could be used to eliminate the **access procedure** calls.

The advantage of solving problems like these by means of direct data visibility is that it is easy to implement and does not imply great complexity in the Mascot **building** process. Furthermore, where **activities** are scheduled for execution in a co-operative manner (see Section 4.4), direct manipulation of **IDA** data need present no threat to their integrity. Under a regime of pre-emptive scheduling safe access may be achievable by very careful programming.

The use of direct data visibility should only be considered, however, where the conventional Mascot methods are incapable of achieving the required results. Where it is adopted it should only be used in a disciplined manner. Direct visibility involves extending the traditional sole responsibility of the **IDA**, in respect of data integrity and propagation, to a group of several components. The scope of this collective responsibility needs to be be well defined and should be limited as far as possible by the use of

qualifiers as discussed below.

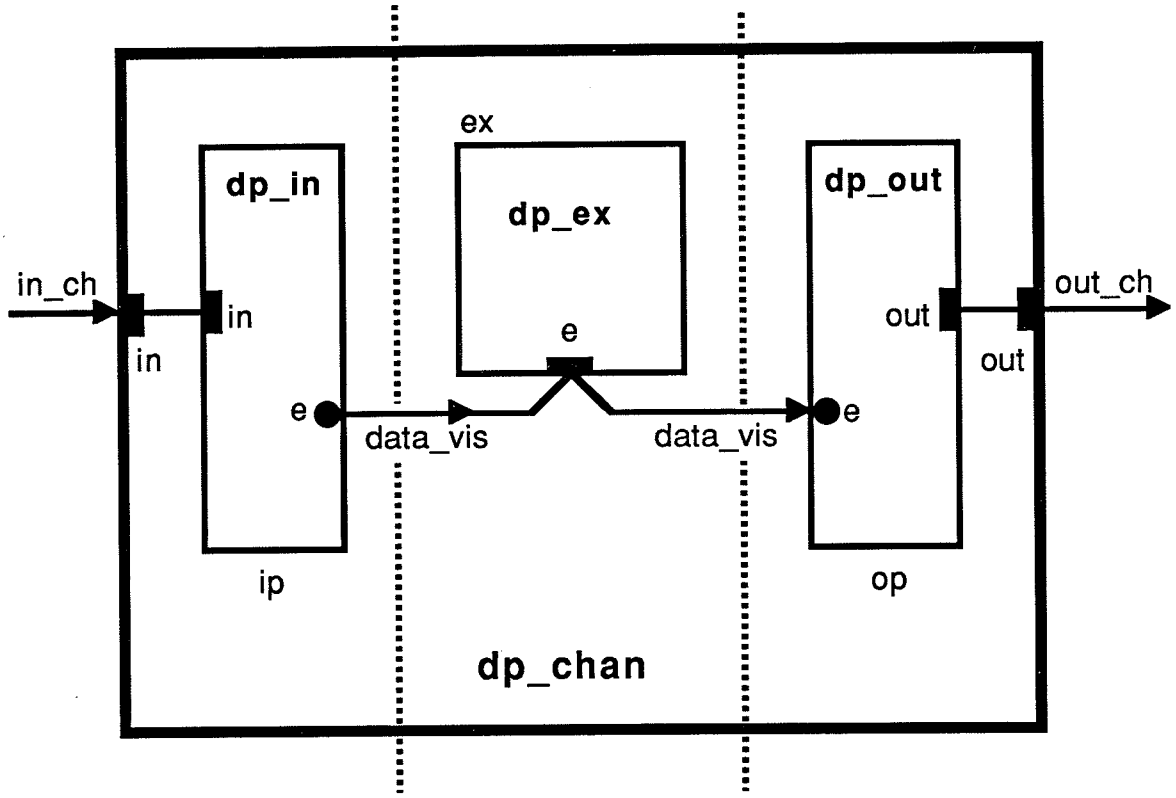
The programming system in use for a particular application may provide an alternative approach to the problems of efficiency and code and data location. One of the following techniques, where available, is generally to be considered preferable.

1. An option provided by a number of modern compilers allows a procedure body to be included, inline, in the object code at the point of call and, subsequently, optimised. Where such a facility is available it might be applied to calls of an **access procedure**, hence achieving an improvement in performance similar to that obtainable through direct visibility. The feature may also provide a means of appropriately locating the code.

2. The problem of explicitly locating code and data may sometimes be solved by means of compiler segmentation options. Many existing compilers provide facilities for splitting compilation units into segments which may be independently located in memory. In simple cases this may involve division into code and data segments but, if steering directives are embedded in the source text, more complex separation is possible. Mascot **building** software could be developed to exploit such compilation facilities. For example, **IDA** code could be placed in shared memory, it could be duplicated in its entirety in the private memory of each processor which uses it, it could be segmented manually and segments duplicated in private memory as necessary or the segmentation could be performed automatically so as to minimise the duplicated code. These examples would achieve progressively greater efficiency of memory usage in return for progressively greater complexity of the **building** software.

Graphical Representation

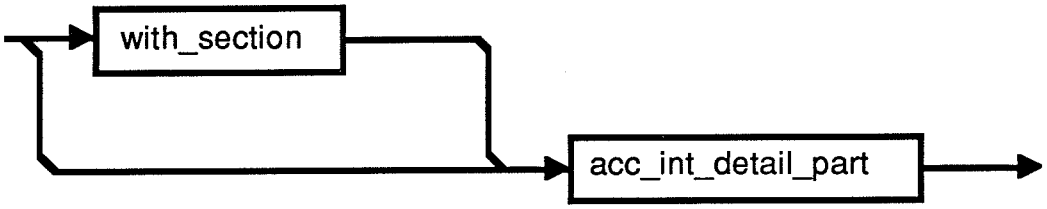
There are no special diagrammatic conventions associated with direct data visibility. The diagram below shows a **composite IDA** in which data in **component** ***ex*** is made directly visible to the other two **components** through the **access interface** ***data_vis***. In this example, for which **modules** are outlined below, ***ip*** and ***op*** are required to be located in the private memory areas of two separate processors while ***ex*** is to be located in a shared memory area. This arrangement is reflected by the broken lines on the diagram.



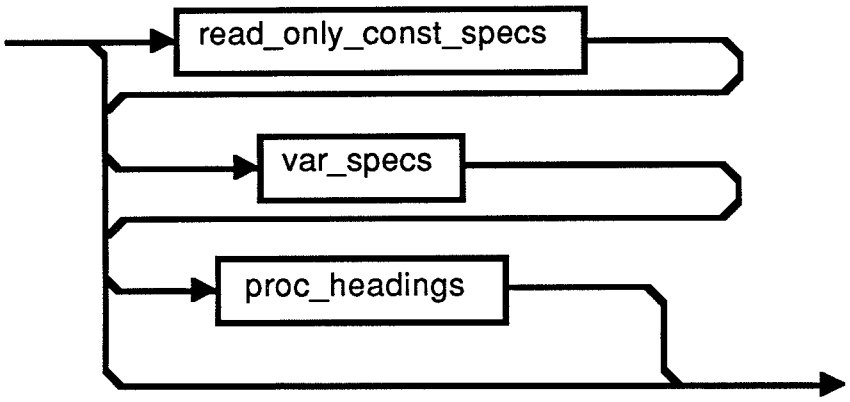
Textual Representation

IDA data may be made directly visible by specifying them in an **access interface** that appears in the **PROVIDES** list of the **module**. Reference to Appendix A shows that the ***specification part*** of a **simple access interface**, in its complete form, is :

simple_acc_int_spec_part



acc_int_detail_part



Thus variables and read-only constants may appear in an **access interface** together with the procedure headings already discussed. It should be emphasised that **interfaces** contain specifications and not declarations. Storage space for **interface** variables and read-only constants, like the coding for **interface** procedures, must be supplied by any **component** which provides a **window** of that type.

Referring to the graphical example given earlier, here is the **module** for the **composite IDA** ***dp_chan***:

```
IDA dp_chan;
   PROVIDES    in : in_ch;
               out : out_ch;

   USES dp_in, dp_out, dp_ex;
   IDA ex : dp_ex;
   IDA ip : dp_in (e=ex.e);
   IDA op : dp_out (e=ex.e);

   in = ip.in;
   out = op.out
END.
```

There are three **access interfaces**, one of which is used internally and involves direct data visibility. It could, for example, be of the form:

```
ACCESS INTERFACE data_vis;
   VAR
      max : CONSTANT integer;    { reading and writing operations }
      data : integer;            { are assumed to be indivisible }
END.
```

The two **interfaces** provided by ***dp_chan*** are conventional in containing only procedure specifications. They are outlined in sufficient detail for present purposes below:

```
ACCESS INTERFACE in_ch;
   PROCEDURE write( item : integer );
   .
END.

ACCESS INTERFACE out_ch;
   FUNCTION read : integer;
   .
END.
```

For the purposes of explanation, the following examples are expressed in a Pascal like notation. **Template** ***dp_ex*** is given in full. Notice the use, in the design representation language, of **ACCESS VAR** by analogy with **ACCESS PROCEDURE**. ***max*** has been rendered as a read-only constant whose value is set in the **IDA template** in order to demonstrate the facility. In practice it would probably

be a **template constant**. The equivalence statements are not strictly required as name identity gives unambiguous correspondence.

```
IDA dp_ex;
   PROVIDES e : data_vis;

   ACCESS VAR
      max : CONSTANT integer := 50;
      data : integer;

      e.data = data;
      e.max = max
END.
```

Only those parts of **template *dp_in*** which are relevant to the present discussion are included in the **module** below. **Access procedure *write*** places input values directly into the internal store ***data*** of **IDA *ex*** which is located in the shared memory region.

```
IDA dp_in;
   PROVIDES in : in_ch;
   REQUIRES e : data_vis;

   ACCESS PROCEDURE write (item : integer);
   BEGIN
      .
      e.data := item;
      .
   END.

   in.write = write
END.
```

The **template *dp_out*** provides **access function *read*** at its **window *out***. This removes values directly from the store in ***ex***.

```
IDA dp_out;
   PROVIDES out : out_ch;
   REQUIRES e : data_vis;

   ACCESS FUNCTION read : integer;
   BEGIN
      .
      read := e.data;
      .
   END.

   out.read = read
END.
```

Finally, to assist in maintaining integrity when using direct data visibility, consideration should be given to the use of the qualifiers SINGLE, READ_ONLY and IDA_ONLY (see later section on this topic).

The qualifier SINGLE, associated with an **IDA window** specification, indicates that only a single thread of execution may pass through this **window**. This effectively limits the scope of any variables directly visible at the **window**. The READ_ONLY qualifier can be used to ensure that externally visible variables are not altered by direct access via that **interface**. The effect of the IDA_ONLY qualifier associated with an **access interface** is to restrict its use to the **ports** and **windows** of **IDAs**. As a result, responsibility for the integrity of visible data is limited to the **components** of a **composite IDA** or equivalent **network**.

2.16 QUALIFIERS

Description

This feature is not part of the mandatory subset of the Mascot definition.

Mascot provides a rich set of facilities for representing the structure of concurrent software. The use of these facilities is constrained in various ways in the interest of maintaining the consistency of the design and the integrity of the resultant software. Such constraints, built into the definition, apply generally to all Mascot systems. However, it will sometimes be found desirable to introduce additional, more severe constraints which apply locally to particular aspects of a design. The concept of a **qualifier** has been devised to meet this requirement.

Examples of Mascot entities to which **qualifiers** may be applied are **ports**, **windows**, **interfaces** and their components. The effect produced by adding a **qualifier** might be to impose or relax some form of constraint on network connectivity, to limit the operations which may be applied to shared data, to indicate (in the textual form of representation) the direction of data flow, to limit the use of selected **context** facilities to certain **classes** of **template** or to influence the action of the compilation system.

It is open to the implementors of a Mascot development environment to provide support for any **qualifiers** that are considered useful for the expected applications. For each **qualifier** that is supported it is required that the following be defined:

- the name of the **qualifier**
- the purpose of the **qualifier**
- where the **qualifier** may be placed and how it is represented
- the effect of the **qualifier**
- whether placing one **qualifier** requires the placement of other similar or complementary **qualifiers** and, if so, what the rules are
- any other information thought to be relevant such as whether violation of the rules constitutes an error or merely results in a warning

Some examples showing how this information might be presented are given later in this section.

Graphical Representation

Qualifiers do not normally appear on Mascot diagrams.

Textual Representation

As indicated above, the representation of **qualifiers** in **modules** is implementation dependent. It is, however, suggested that qualified **windows** and **ports** might take the following form:

REQUIRES a, b/READ_ONLY, c : get;
PROVIDES d/SINGLE : put;

where the proposed significance of these **qualifiers** is described below.

Examples

The sample **qualifiers** given below are those which were suggested during the discussion of this concept in the course of development of the Mascot 3 definition. They are divided into categories each of which is now briefly introduced.

Connectivity Constraints

The default connectivity requirement for a **window** is that at least one **port** must be connected, and possibly several. Use of the **qualifiers** in this category has the effect of adjusting these requirements.

Data Access Constraints

The **qualifiers** in this category place limits on access to variables made directly visible via an **access interface**.

Data Flow Indicators

These **qualifiers** are used to indicate the direction of data flow with respect to **ports** and **windows**. This is information which, in the mandatory subset of Mascot, is expressible only in the graphical form of a design.

Context Qualifiers

In this category, the **qualifiers** enable the designer of a context interface (see Section 4.1) to determine which facilities are available to which **classes** of **templates**. In some cases the limitation is to particular features of a **template class**.

Code Generation Constraints

The **qualifier** in this category can be used as a possible alternative to making variables visible and directly accessible via **windows**. For a more detailed discussion of this point see Section 2.15 of the Handbook.

CONNECTIVITY CONSTRAINTS

Name	Place	Description
SINGLE	**window** of **IDA** or **server**	**Purpose:** to indicate that the designer of the **IDA** or **server** has optimised the implementation in such a way that only a single **activity** may use that **window**. **Effect:** Only one **activity port** may be connected to this **window**. **Additional Information:** Care is needed in applying this constraint where **composite** design entities are involved. In particular the 'singleness' of a **window** must be passed out to the **windows** of the enclosing **composite** structure and the number of internal **port** to **port** connections within enclosing **subsystems** must be determinable.
OPEN	**window**	**Purpose:** to indicate that a **window** need not necessarily be used in an operational **network**. It may, for example, have been included purely for testing purposes. **Effect:** The ENROL operation will not issue a warning if a **window** qualified as OPEN has no **port** connected to it. **Additional Information:** OPEN **windows** may have zero or more connections. **Windows** marked as both OPEN and SINGLE may have zero or one **port** connected.

DATA ACCESS CONSTRAINTS

Name	Place	Description
READ_ONLY	**port** or **window**	**Purpose**: to prevent the values of variables, visible through the **window**, being altered via **port(s)** connected to the **window**. **Effect**: Any attempt to write data through a READ_ONLY **port/window** connection is identified as an error when the ENROL operation is applied to the **template** concerned. **Additional Information**: Where a **window** is qualified as READ_ONLY, it should only be connected to a similarly qualified **port**. However, a READ_ONLY **port** may be connected to any **window**. In order to limit the scope of cross-checking, it may be desirable to insist that the READ_ONLY qualifier must appear on all of the **ports** and **windows** associated with the **path**.
IDA_ONLY	**window**	**Purpose**: to ensure that only **ports** of **IDA**s are connected to the qualified **window**. **Effect**: Any attempt to connect the **port** of an **activity** (directly or indirectly via intervening **subsystem** boundaries) to an IDA_ONLY **window** will result in an error being identified when the ENROL operation is applied to the **network** which contains such a connection.

DATA FLOW INDICATORS

Name	Place	Description
IN	**window** or **port**	**Purpose**: to indicate the direction of data flow. **Effect**: The direction of data flow may be expressed in the textual form of a **template**.
OUT	**window** or **port**	**Purpose**: to indicate the direction of data flow. **Effect**: The direction of data flow may be expressed in the textual form of a **template**.
IN_OUT	**window** or **port**	**Purpose**: to indicate the direction of data flow. **Effect**: The direction of data flow may be expressed in the textual form of a **template**.

CONTEXT QUALIFIERS

Name	Place	Description
SERVER_ONLY	context **Interface** or component of context **Interface**	**Purpose:** to make qualified facilities available only to **servers**. **Effect**: The use, in a **template** which is not a **server**, of facilities qualified in this way will result in an error being identified when the ENROL operation is applied to the **template**.
HANDLER_ONLY	context **Interface** or component of context **Interface**	**Purpose:** to make qualified facilities available only to **handlers**. **Effect:** The use, in a **template** which is not a **server**, of facilities qualified in this way will result in an error being identified when the ENROL operation is applied to the **template**. Further, when a **server template** is enrolled, only the code of any **handlers** inside it will be allowed to use the qualified facilities. Any other uses will result in an error being identified and cause the ENROL operation to fail.
NOT_HANDLER	context **Interface** or component of context **Interface**	**Purpose:** to prohibit the use of qualified facilities by **handlers**. **Effect:** The use, by a **handler**, of facilities qualified in this way will result in an error being identified when the ENROL operation is applied to the **server template** which contains it.

IDA_ONLY	context **Interface** or component of context **Interface**	**Purpose:** to make qualified facilities available only to **IDA**s. **Effect:** The use, by any **template** which is not an **IDA**, of facilities qualified in this way will result in an error being identified when the ENROL operation is applied to the **template**.
ACTIVITY_ONLY	context **Interface** or component of context **Interface**	**Purpose:** to make qualified facilities available only to **activities**. **Effect:** The use, by any **template** which is not an **activity**, **root** or **subroot** will result in an error being identified when the ENROL operation is applied to the **template**.

CODE GENERATION CONSTRAINTS

Name	Place	Description
INLINE	**port** or **window**	**Purpose:** to avoid the overhead involved in calling **access procedures**. **Effect:** The compilation system is forced to copy the code of the **access procedures**, available at a qualified **window**, into the code of a calling **template** at the point of call. **Additional Information:** It would be reasonable to provide a facility whereby the INLINE qualifier may be suppressed by means of an IGNORE_INLINE option of the BUILD operation.

CHAPTER 3

DEVELOPMENT FACILITIES

3.1 STATUS PROGRESSION

Introduction

An essential feature of a Mascot development environment is a **database** capable of containing a collection of **modules** from which, together with their derived products, application software may be created. As will be demonstrated later, Mascot **modules** are so defined as to facilitate the progressive, incremental elaboration of a design. In support of this, the **database** accords formal recognition to the attainment of certain important progress milestones in the development of a **module**. A **status** value, associated with each **module**, provides a measure of the level of recognition attained and consequently of the **module**'s fitness for use. This value reflects not only the progress made in defining the **module** itself but also the state of other **modules** to which direct or indirect reference is made. It is assumed throughout this section that **status** values are maintained automatically in the **database** by the Mascot development environment but a manual recording procedure would also be possible. A development environment may provide facilities to display the **module status** and possibly the inter-**module** dependencies.

As **status** progression is closely associated with **module** structure this is reviewed first.

Module Structure

Reference to the syntax diagrams in Appendix A shows that every Mascot **module** begins with a **name part** which defines its **class** and gives the **template** or **specification** it represents a unique name. This is followed by a **specification part** whose purpose is to define that part of the **module** which needs to be known when it is used by other **modules**. It establishes the external view of the **module**.

For **specifications**, the **specification part** consists of the detail of the **module** together with a statement of its dependency on other **modules**. Such external dependencies occur only in **simple specifications** and are limited to the importation of data-type definitions from other **specifications**. They are expressed in the form:

WITH definition module list {***importation of data-types*** }

For **templates**, the **specification part** consists of the information required for **components** of that type to be included in a **composite template**. This information includes several varieties of external dependency. Connections between **objects** are expressed in their **templates** by means of references to **specifications**. There are two different kinds of connection which are expressed in this manner and in each case the active partner in the transaction is distinguised from the passive partner. First there are those which convey network interactions and are concerned with communication between

ports and **windows**. These dependencies take the form:

REQUIRES port list { ***network interactions*** }
PROVIDES window list

Second there are those which convey sequential program interactions between the individual, separately developed components of a single thread of execution:

NEEDS subroot interface list { ***sequential program interactions*** }
GIVES subroot interface list

A third form of external dependency, closely related to the second category above, occurs in the case of a **template** which describes a collection of shared **library** facilities. The set of interactions which this passive entity makes available are expressed as:

GIVES library interface list { ***provision of library services*** }

The final possible element of a **specification part** is the **template constant** expressed as:

CONSTANT template constant list { ***template constant*** }

Templates (but not **specifications**) possess a further section known as the **implementation part** which defines the internal details of the **template**. For **simple templates** this defines the program. For **composite templates** it defines the **components** together with their connections and **template constant** values. **Simple templates** may import data-type definitions directly from **specifications** and may make use of facilities provided by **library modules**. These dependencies are expressed in the form:

WITH definition module list { ***importation of data-types*** }

LIBRARY library interface list { ***use of library services*** }

A **composite template** depends on those **templates** from which its **components** are derived. These are listed:

USES template list { ***templates for components*** }

Appendix C (Summary of Keyword Usage) lists all the inter-**module** dependencies in terms of the representation language keywords used to express them and distinguishes between those associated with the **specification** and **implementation parts** of a **module**.

Status Values

The **status** value of a **module** directly reflects the state of completion and validation of the three sections of its structure. There are thus three primary **status** values, **registered**, **introduced** and

enrolled, the attainment of which record the progressive successful validation of the **name**, **specification** and **implementation parts** of the **module**, respectively. The operations, computer-assisted or manual, which establish these **status** values are REGISTER, INTRODUCE and ENROL.

A **module**'s primary **status** value may be qualified to reflect the **status** of the other **modules** which belong to the **specification** and **implementation** dependency trees of which it is the root. Thus, a **module** whose **specification part** is present and correct, but which has specification dependencies still at **registered status**, may be raised to the **status** of **partially introduced**. It qualifies to become **fully introduced** when all its specification dependencies have achieved this **status**.

For a **composite template** there is a **status** value of **partially enrolled**. This **status** can be achieved when all three sections of the **module** are present and correct and all its specification and implementation dependencies are at least **partially introduced**. As will be demonstrated in the example below, the **partially enrolled status** conditions enable the system designer to complete a network design before considering the interface and algorithmic details of the design.

The formal rules governing the achievement of these **status** values are summarised in a table at the end of this section. Their application is illustrated below. In the example, **status** progression is shown as an orderly increase in **status** from **registered** to **enrolled**. In practice, facilities must be provided to support iteration and the consequent backtracking.

Example of Status Progression

The following example illustrates how a **subsystem** might be developed progressively. The detailed features of the Mascot development environment employed in this example are not mandatory to the Mascot definition. The **subsystem** chosen for illustration is the one called ***subsys_4*** in Section 2.1 of the Handbook.

The **status** progression commands may be used to control the development of a **system** by a team of people. We shall assume, as the starting point for our example, that **subsystem** ***subsys_4*** has been **introduced** and has achieved **partially introduced status**. This implies that the **access interfaces** required and provided by ***subsys_4*** have been **registered** and that the only information known to the **Mascot database** about ***subsys_4*** is the following

```
SUBSYSTEM subsys_4;
   PROVIDES gw4 : get;
   REQUIRES rp4 : rec; otp4 : out; tp4 : trans;
```

Our task in the example is to develop ***subsys_4*** and we shall define a series of states through which the **subsystem** and its constituents must pass. These states will be visible in the **Mascot database** as a result of applying the **status** progression commands and can thus be used to control, record and monitor progress.

There are three significant states which arise during the development of ***subsys_4*** :

1 Design structure recorded for ***subsys_4***.

2 Ready to start implementation of **templates** from which the **components** of ***subsys_4*** are derived.

3 Completion of implementation of all **templates** used by ***subsys_4***. This implies that ***subsys_4*** may be built.

State 1 is reached when ***subsys_4*** is **partially enrolled**. This implies that the **templates** for all of its **components** have been **introduced** and have achieved **partially introduced status**. State 2 is reached when all the **simple templates** for the **components** of ***subsy_4*** have achieved **fully introduced status** and state 3 is reached when ***subsys_4*** is **fully enrolled**.

Reaching State 1

To move from our starting point to state 1, it is necessary to design the internal structure of ***subsys_4***. This will most probably be represented graphically first and, if a graphics tool is available, could be recorded directly in this form. In such a circumstance the graphics tool can, via interactions with the user, determine which additional **modules** need to be **registered** and **introduced** in order to render ***subsys_4*** capable of achieving **partially enrolled status**. However if a text based design checking tool is used, then the next step would be to transcribe the graphical representation of ***subsys_4*** into its textual equivalent, thus:

```
SUBSYSTEM subsys_4;
   PROVIDES gw4 : get;
   REQUIRES rp4 : rec; otp4 : out; tp4 : trans;
   USES pool_1, chan_1, a_temp_1, a_temp_2;
      POOL p1 : pool_1;
      CHANNEL ch : chan_1;
      ACTIVITY a1 : a_temp_1 (fp = ch.fw, tp = tp4, pp = p1.pw);
      ACTIVITY a2 : a_temp_2 (sp = ch.sw, otp = otp4, rp = rp4);
      gw4 = p1.gw
END.
```

From this (or directly from the graphical representation) it can be seen that in order to **enrol** this design, it is necessary to have the four **templates** ***pool_1***, ***chan_1***, ***a_temp_1*** and ***a_temp_2*** at **introduced status**. Further, before these **templates** can be **introduced**, it is necessary to

register them and the **access interfaces** upon which they depend.

The specifications for the four component **templates** of ***subsys_4*** are:

```
ACTIVITY a_temp_1;
   REQUIRES fp : fetch; tp : trans; pp : put;

ACTIVITY a_temp_2;
   REQUIRES sp : send; otp : out; rp : rec;

CHANNEL chan_1;
   PROVIDES sw : send; fw : fetch;

POOL pool_1;
   PROVIDES pw : put; gw : get;
```

It can be seen that the four component **templates** between them make use of seven **access interfaces** of which four are visible at the **subsystem template**. These four will already have been **registered** to allow ***subsys_4*** to achieve **partially introduced status**. Therefore we need to **register** the three additional **interfaces** ***fetch***, ***send*** and ***put***. Having **registered** the four **templates** and the three **access interfaces**, we can **introduce** the **templates** and thereby submit them for formal checking of their **specification parts**.

The INTRODUCE operation first checks that the **name parts** of the **modules** have not been changed since they were **registered** and then checks the **specification part**. The **specification part** must itself be syntactically legal and any other **modules**, to which the **modules** being **introduced** refer, must be at least at **registered status**. These are the minimum requirements for the INTRODUCE command to succeed and, in our example design, these minimum requirements are met. Therefore the four **templates** qualify for **partially introduced status**.

The INTRODUCE operation will also check whether the preconditions for **fully introduced status** are satisfied. At this stage in the development of our example these conditions are not met because the **interfaces** are not yet **introduced**. Therefore the **status** achieved is **partially introduced**.

It is now possible to **enrol** the design structure for ***subsys_4***. This operation checks that the **specification part** has not been changed since the **module** was **introduced** and that the design details represent a consistent use of the **templates** and **interfaces** involved. As a result, ***subsys_4*** will achieve **partially enrolled status**. State 1 has now been reached.

<u>Reaching State 2</u>

In order to reach state 2, it is necessary to have all seven of the **access interfaces** specified in detail and for them to be raised to **introduced status**. For the three internal **access interaces** we, as the nominated design authority (see section 5.1), can proceed immediately. For the four external **access**

interfaces we must liaise with other groups, who are developing **modules** which also use these **access interfaces**, and with the nominated design authority for them.

When the internal details have been agreed, they are recorded as the texts for the **access interface modules** and any **definition modules** found to be necessary. These details are then submitted for checking and entry into the **Mascot database** via the INTRODUCE command. The sequence of commands is as follows: REGISTER all new **definition modules**; INTRODUCE **definition modules**; INTRODUCE **access interface modules**.

Assuming that these operations are succssful we now have all the **access interfaces** for ***subsys_4*** at **fully introduced status**. In order to reach state 2 though, we have to bring the **Mascot database** up to date by re-**introducing** the four **simple templates** ***a_temp_1***, ***a_temp_2***, ***pool_1*** and ***chan_1*** and finally re-**enrolling** ***subsys_4***.

Reaching State 3

The action necessary to reach state 3 is to provide the implementation details for each of the **simple templates** used for the **components** of ***subsys_4***. This is a programming task and each **module** can be assigned to a different individual within the team who will then add the details and submit the assigned **module** for **enrolment**. When all the implementation details have been provided and checked to be consistent with the **fully introduced interfaces** by the ENROL command, we have almost reached state 3. It only remains to re-**enrol** ***subsys_4*** so that it can be related, in the **Mascot database**, to the **fully enrolled simple templates**. Then ***subsys_4*** can achieve **fully enrolled status** and our task is completed.

This state formally constitutes the end of the implementation phase for ***subsys_4*** and signals the start of the testing phase for that **module**. Depending upon the testing strategy adopted, there might well be other **subsystems** developed specifically to provide test harnesses for ***subsys_4***. The development of these **subsystems**, and indeed the test **system** required to execute them, will follow similar lines to that described and use the same commands.

More Sophisticated Status Progression Commands

The **status** progression commands as described here are very simple and operate purely on one **module** at a time, although they do require checks on the **status** of other **modules**. It is envisaged that more sophisticated versions of the INTRODUCE and ENROL commands could be provided. These would be especially useful for large **systems** particularly during the later stages of integration and during maintenance.

In the above example, the requirement to re-apply the INTRODUCE and ENROL commands, as more details of the design are added, has been identified. Doing this manually is possible but tedious and error prone. If one of the commands was omitted in error, this could result in a **system** being built which failed to incorporate some recent **module** amendments. The possibility of this happening increases as the size of the **system** being developed increases. Therefore, more sophisticated forms of the ENROL and INTRODUCE commands are defined which automatically seek out any later versions of **modules** and re-apply the appropriate command. Hence the extended form of the ENROL command, possibly invoked as: ENROL subsys_4 FULLY, would be interpreted to mean re-**enrol** ***subsys_4*** and any **modules** which it directly or indirectly depends upon. Thus this leads to a recursive application of the ENROL command.

Similarly the extended form of the INTRODUCE command, possibly invoked as INTRODUCE FULLY, would result in recursive application of the INTRODUCE command.

A more far-reaching extension of the ENROL operation would allow re-**introductions** as well as re-**enrolment**. This interpretation may be of value when an **interface** change is made either late in the development of a **system** or during the maintenance stage. It provides a simple way of bringing all the **components** of a **system** to consistency with the latest issue of the design.

Status Conditions			
Operation	**Status to be achieved**	**Module Class**	**Preconditions**
Register	Registered	All	Name part defined and legal No other module with same name
Introduce	Partially Introduced	All	Registered preconditions satisfied Specification dependencies Registered Specificatiopn part defined and legal
	Fully Introduced	All	Partially Introduced preconditions satisfied Specification dependencies Fully Introduced
Enrol	Partially Enrolled	Composite Templates	Partially Introduced preconditions satisfied Implementation dependencies Introduced Implementation part defined and legal
	Fully Enrolled	Simple Template	Fully Introduced preconditions satisfied Implementation dependencies Fully Introduced Implementation part defined and legal
		Composite Templates	Partially Enrolled preconditions satisfied Implementation dependencies Fully Enrolled

3.2 SYSTEM BUILDING

Introduction

Building is the term used in the Mascot definition to describe the stage of development which, starting from a **fully enrolled system template**, produces a representation of the network in an executable form. The process which achieves this must take into consideration the target configuration for which the application is to be built: the number and type of processors to be employed, the accessibility of memory from each processor and any special requirements for interfacing with devices. In view of this strong target dependence and the wide variation in the range of **building** facilities required for different kinds of application, the Mascot definition does not legislate on the precise form which the facilities should take or on the linguistic support which a Mascot development environment should provide in this area. Rather, in this section, an attempt is made to discuss all the factors which need to be considered and to recommend and justify some preferred modes of working practice.

Building Strategies

The ultimate objective of **building** is the construction of a complete operational system. However to facilitate development, particularly during the phase when individual modules are being tested, it is desirable to be able to **build** test **systems** which incorporate only part of the final **system**. This can be achieved by means of a **system template**, specially created for the purpose, which encapsulates the components to be tested. By associating such test **systems** with the test procedures and results, the development process can be documented reliably and specific test exercises can be repeated should the need arise.

In general, the object of a test will be a subsidiary **network** extracted from the complete system design and possessing, in consequence, **components** with unconnected **ports** and **windows**. A test **system** must therefore contain additional **components** to supply these missing connections. Thus, the test **network** may be completed by means of a dedicated test harness capable of supplying any required input and of recording and possibly checking any generated output. Alternatively, input and output could be achieved by direct connection to external devices such as a set of files on a host computer. In this latter case the additional **components** would consist of **servers**.

While it may be acceptable for a small test **system** to be re-**built** from scratch each time amendments are made to the **modules** from which it is derived, this is likely to be a rather laborious and time consuming procedure for the large **systems** typical of Mascot applications and even for the bigger test **systems** employed during the integration phase of development. The time taken to **build** a **system** image may be reduced if a number of the component **subsystems** have previously been separately **built**. The **context** software for a particular computer is an obvious candidate for being made available as a partially

built image. However, this approach is more suited to some target architectures than others. It is, for example, particularly useful when **building** for a multi-processor target where the constraints are uniform. But where a memory mapping scheme is employed in the target, the **builder** requires an overall view of the **system** in order to achieve optimal results and so **building** speed, in this case, might have to be paid for in reduced efficiency.

During module and integration testing, the same **system** may be **built** and re-**built** many times. A sophisticated **builder** might detect that, between one **build** operation and the next, only a few components had changed and so could save time by re-**building** as little of the **system** as possible. A more limited **builder** might achieve the same result with a little assistance from the user.

Linguistic Support for Building

A simple constructional task would be that of **building** a **system** for a target consisting of a single processor, of a pre-determined type, which has access to sufficient memory, all of a uniform type, and making use of a predefined **context**. This could be performed by a dedicated **builder** supplied with no more data than the identity of the **system template** and the values of any **template constants**. This information could readily be accommodated as a set of parameters to a BUILD operation:

```
BUILD total_sys, 100, 50
```

It would be preferable however, even in so simple a case, for the data to be presented in a BUILD **module** to which reference could be made in initiating a BUILD operation:

```
BUILD build_mod
```

This approach allows the image to be brought under configuration control so that its regeneration can be guaranteed. The range of information held in the BUILD **module** can be extended for use by more powerful **builders**. Where a choice of **contexts** and processor types exist, for example, the selections could be specified in the **module**.

To cater for more complex targets more **build**-time information must be supplied and a decision must be made on whether to hold it all in the one data **module** or to introduce others. Suppose, for example, that the **builder** supports target configurations with disjoint memory blocks or non-homogeneous memory with differing speeds or modes of access. The size of each memory block, its address range and its properties must be made available to the **builder** together with instructions as to where the system components are to be located among the various memory blocks. Placing all this information in a single BUILD **module** would necessitate its replication for each **system built** for the same target configuration and so it might be preferable to use a separate (TARGET) **module** to hold details of the target.

Separation of the **build**-time data in this particular way would still prove inflexible where there was a requirement to **build** images of the same **system** for several target configurations. A possible solution would be to define the target in software terms, such as computers and memory regions, and to define the location constraints in terms of these regions. The target could then be defined in hardware terms, such as processors and memory blocks, and a correspondence given between the two views. For maximum flexibility, this information could be divided between four separate **modules** each belonging to a distinct **class**:

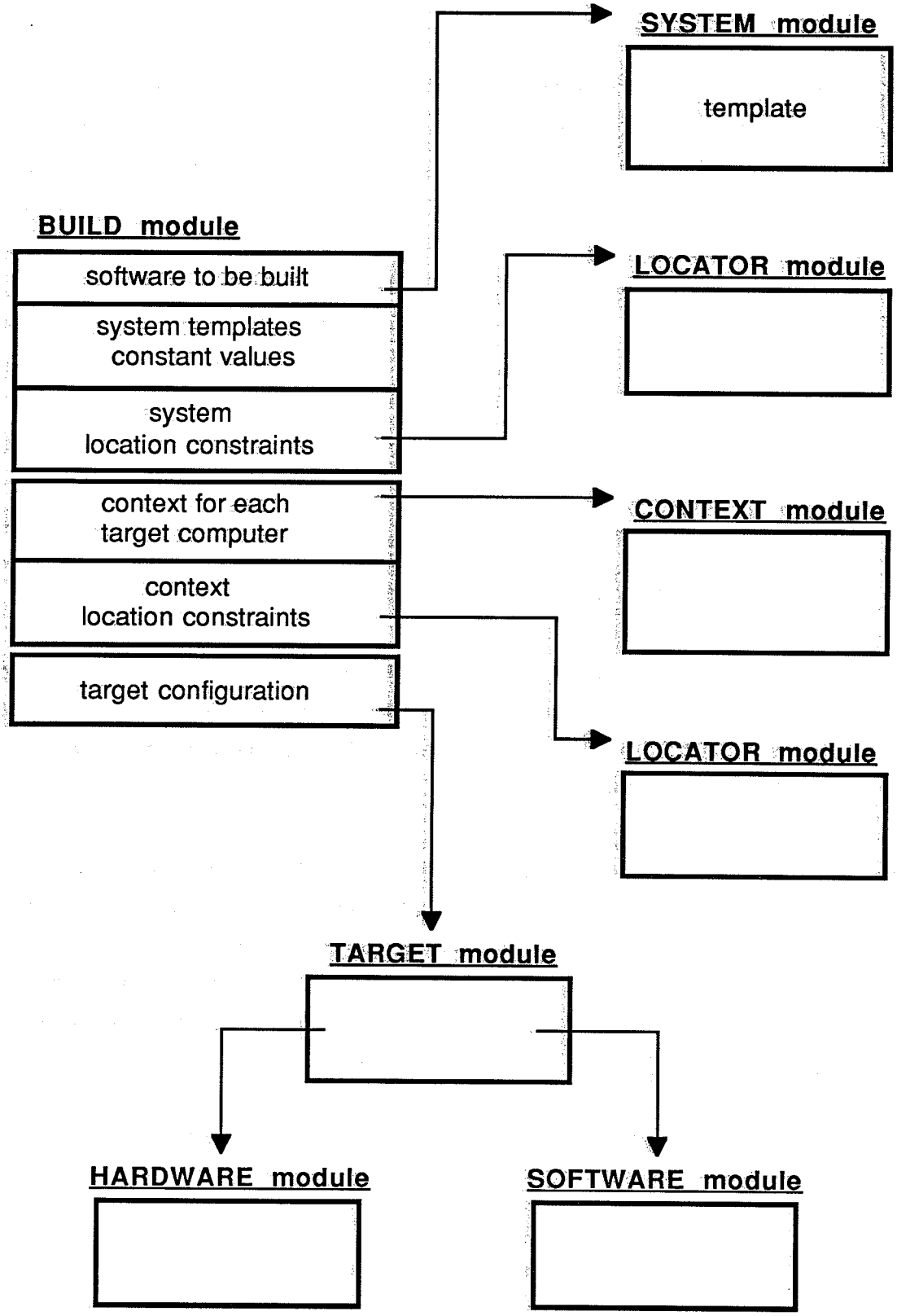
BUILD
TARGET
HARDWARE
SOFTWARE

The location contraints could then be spread among any number of LOCATOR **modules**.

It may be that devices in the target configuration are memory mapped. In such a situation, communication with the devices may be through pre-determined normal or special memory locations. If the necessary device access information is not embedded in the applications software then it must be supplied via the **builder**, perhaps by the use of **template constants**.

Further **build**-time data is required to support target configurations consisting of multiple processors with shared memory. The **builder** needs to know which memory blocks are private to a processor and which can be or which are to be shared by more than one processor. It may also need an indication of which components are to be allocated to each processor. Where the target processors are memory mapped, the **builder** may require guidance as to how each logical memory region is mapped into a physical memory block.

In its full generality then, the BUILD **module** might supply information to the **builder** as indicated in the diagram below:


SYSTEM module
template
BUILD module
software to be built
system templates
constant values
system
location constraints
LOCATOR module
context for each
target computer
context
location constraints
CONTEXT module
target configuration
LOCATOR module
TARGET module
HARDWARE module
SOFTWARE module

3.3 DEVELOPMENT CONFIGURATIONS

Introduction

In discussing, in this section, a range of hardware configurations appropriate to the development of Mascot software, a distinction is made between a host system, on which the software development takes place, and the target system, on which the operational software is to be executed. The different host-target combinations which are encountered in practice vary in the degree of similarity which exists between the two systems.

At one extreme, the host and target may be the same system or the host may combine a processor which is identical to that of the target with a subset of the target's peripherals. Alternatively, because development involves requirements not relevant to operation and because the target peripherals are frequently of an exotic nature, the host peripherals may be different to those of the target while the processors are the same.

In other arrangements, the processor in the host system may possess a similar instruction set and number representation to those of the target but may differ in other respects such as the manner in which input/output is performed. Sometimes the host and target have a common number representation while their instruction sets differ. Finally, development may take place on a host system whose processor has nothing in common with that of the target at all.

Commissioning Configurations

The hardware configurations mentioned above may be employed to support formal or informal execution of the software for the purpose of quality assurance and for the diagnosis and correction of errors. Where the host and target processors are identical, the validity of testing depends only on considerations, including those discussed below, which relate to the run-time environment. The four most commonly encountered configurations in which host and target systems differ are examined here in order to highlight their relative advantages and disadvantages.

Native Code Execution on the Host

In the first configuration, tests may be executed on the host in its own native code. Results may be obtained rapidly in advance of the target system becoming available for use. Since, in general, the host can handle much larger programs than the target, it is possible to test the complete system in the presence of dedicated test software such as simulators and scenario generators. Good diagnostic facilities can be provided and all the host peripherals can be made accessible to the tests. In many host systems the passage of system time can be controlled though usually in a relatively crude manner.

Where the host processor differs from that of the target, especially in respect of wordlength, the range and accuracy of the arithmetic results obtained in tests may not represent a true indication of the performance to be expected on the target. Other aspects of the system may not be satisfactorily exercised. For example if, as is common, only co-operative scheduling is available on the host the testing of mutual exclusion and cross-stimulation within **IDA**s is not wholly adequate. Some parts of the code, such as that involved in handling interrupts, usually have to be omitted from testing altogether.

It is important that host and target should be compatible in respect of high level program statements which must, of course, be recompiled for execution on the target when testing on the host has been completed.

Emulation on the Host

In this second approach, source code compiled for the target is interpretively executed on the host. Since there is only one compiler, ie that for the target, there is no question of incompatibility between host and target. The results will be accurate and reliable provided the emulation is accurate. This method possesses even better diagnostic capabilities and access to all host peripherals than that discussed above. In particular it provides an opportunity to increase the level of run-time checking and validation. Checks can be performed, for example, for arithmetic overflow, for the use of uninitialised memory locations and for the corruption of code.

The principal disadvantage is that execution time, from 10 to 1000 times longer than on the target, limits the size of system that can be tested in this way and frequently excludes all real-time aspects from the tests. In order to achieve as high an execution speed as possible, it may be useful to exclude the Mascot kernel from the emulation. The simulation of interrupts is difficult. Control of the passage of system time is difficult but to expend effort in achieving it will usually be cost effective.

Execution under Host Control

A third, commonly employed, configuration involves the execution of tests on the target under the control of, and monitored by, the host via a direct link. The validity of the results may be adversely affected by timing distortion arising from the need to handle communications with the host. A real Mascot kernel can be used although this may contain facilities, such as monitoring, which are not to be included in the final operational software. Provided that the target system is equipped with a full set of operational peripherals, all of the software can be tested. Only a target compiler is required and there is no need for source code compatibility with the host. Potentially the diagnostic capabilities are as good as in the previously discussed arrangements

The disadvantages are first that, while access to all the host peripherals can be provided, it is frequently of limited bandwidth. Second, it is not normally possible to control the passage of system time so as to compensate for the additional overheads attributable to the presence of test components and to the link to the host.

Stand-alone Execution on the Target

In the fourth, and final, configuration discussed here tests are carried out on a free standing target utilising only the peripherals connected to that system. Such an approach has none of the advantages of the host-linked arrangement described above but interference from the link is eliminated. It is unlikely that the target system provides a suitable environment for testing purposes and so tests will be laborious and time consuming to set up and carry out. The diagnostic capabilities are likely to be very limited and the available peripherals may consist of no more than a control panel or a keyboard terminal.

Both methods in which testing is performed on the target encounter further complications when the target consists of multi-processors which have some shared memory. In this case, the target system may not, during some phases of testing, be fully equipped with processors.

The Software Environment for Test Execution on the Host

It is normally desirable, in most development environments, to perform at least some of the testing on a host system. Among the many reasons for this is lack of availability of the target system during the testing phase of a project. This may simply arise from the problem of inaccessibility which is due to the target's location at a remote site and is likely to be exacerbated during the maintenance phase after the system has been installed. Frequently however, the software development team obtains access to the target system only at a very late stage of production and the system may even then not include all the specialised peripherals. Thus testing on a host permits hardware and software development to proceed in parallel.

As was made clear above, in the discussion of test configurations, the target system is likely to contain few of the diagnostic tools which are commonly available on a host. Furthermore, the target may not have the peripherals necessary to support diagnostic output. Shortage of memory may also be a problem. It will rarely be possible to find storage space for test software since all the memory which can physically be accommodated will in practice be needed for the application. If it were not, the space would be used for other equipment.

A further disadvantage of testing on the target arises from the length of time taken to transfer code from the host. Transfer may be via a relatively slow link or, indeed, may be performed through the use of a magnetic medium such as a cassette. A PROM-based target represents an extreme manifestation of this problem involving as it does the transfer of executable code into PROMs. Difficulty may also be encountered in transferring test data and results to and from the target. Even a direct link may be inconveniently slow for this purpose and may result in excessive distortion of the real-time aspects of system performance.

The degree of compatibility between host and target machines is the most significant factor affecting the usefulness of testing on the host. Unless the two processors are identical, the degree of compatibility is

determined principally by the choice of language and compilers and by the level of support for compatibility provided by the development environment. Where the code of at least the majority of the application **templates** can be executed on either the host or the target, with similar results, then host testing is likely to be useful. It is quite feasible in this case to test almost all the software, up to and including the full system, on the host. Those small sections of program which cannot be dealt with in this way can be tested after transfer to the target and before proceeding to the integration tests. Some iteration involving a return to unit testing on the host may be implied by the detection of errors during integration.

Where host and target are incompatible for all, or the majority, of the application **templates** the advantages of host testing are lost since the source code must be edited before being transferred to the target. This presents a serious maintenance problem and, in addition, necessitates the tests being repeated on the target system.

CHAPTER 4

RUN-TIME FEATURES

4.1 CONTEXT SOFTWARE

In any software system it is normally possible to distinguish between functions which are part of a particular application and those which are more properly to be regarded as part of the environment in which the software operates. In Mascot, this division is reflected by the distinction between application and **context** software. A Mascot development environment may be expected to offer a collection of supporting services from which software designers can select the set of services needed for a particular application. These may include, for example, the conventional Mascot primitive operations described in Section 4.2. The **context** may also include application specific items such as procedures for controlling peripheral devices. The general principle is that a set of procedures, constants and data-types provided in the **context** for a particular application is implicitly available. The set may be subdivided so as to restrict the use of defined subsets of the **context** facilities to specific **classes** of design entity such as **servers**, **IDAs** or **activities**.

The interfaces which the **context** offers to the application are described in a form which is consistent with the Mascot modularity scheme. The term **context interface** is used to embrace all those **interfaces** which are implicitly available to the application software. The precise form in which it is expressed is implementation dependent but should be generally compatible with the style of the application software **modules**. Its **components**, however expressed, are a mixture of those associated with the **library interfaces**, **access interfaces** and **definitions** described elsewhere.

In a simple case, the **context interface** might be expressed as a combination of a **library interface** and a **definition** which together specify procedures to be called and define data-types for use as parameters. The procedure specifications might include control queue primitives together with a peripheral library:

```
DEFINITION cq_def;
   TYPE
      controlq = ..... ;
END.

CONTEXT INTERFACE con_procs;
   WITH cq_def;
{ Control queue primitives }
   PROCEDURE join( VAR q : controlq );
   PROCEDURE leave( VAR q : controlq );
   .
{ Peripheral library }
   PROCEDURE switch_on_device;
   PROCEDURE switch_off_device;
   .
END.
```

Alternatively, the **context** interface can be expressed in a **composite** form, comprising a number of interfaces. The above example could thus be expressed:

```
LIBRARY INTERFACE cq_prims;
   WITH cq_def;
   PROCEDURE join( VAR q : controlq );
   PROCEDURE leave( VAR q : controlq );
   .
END.
```

The peripheral library specification could then be placed in a conventional form of **library interface**:

```
LIBRARY INTERFACE periph_lib;
   PROCEDURE switch_on_device;
   PROCEDURE switch_off_device;
   .
END.
```

and the **context interface** itself expressed in the **composite** form:

```
CONTEXT INTERFACE context;
   COMPRISES cq_prims, periph_lib;
END.
```

An **access interface** might feature as a **component** of a **context interface** in order to give access to an **IDA** or **server** in the support software. Special, implementation dependent calling mechanisms may also be provided by the **context** through additional **interface** types and qualifiers.

The Mascot definition lays down no firm rules for the expression of **context interfaces**. The above are merely examples of some acceptable forms within the general spirit of the definition. However, procedure names in the **context** must be unique and any implementation must define the **context** facilities provided and the means for **context** extension.

4.2 Synchronisation of Activities

The Use of Implementation Language Concurrency Facilities

Implementations of Mascot development environments prior to the advent of Mascot 3 have invariably supported programming languages, such as Coral 66, which make no provision for concurrency. With the planned extensive use of languages such as Ada, which include concurrency as a feature, this situation may be expected to change. Consequently the Mascot facilities described in this section, which cater for the necessary synchronisation of separate parallel threads of execution, are to be regarded as constituting a model. This model may be implemented directly in a conventional programming language, as in the past, or mapped onto equivalent features in a concurrent language.

In any implementation of a Mascot development environment for which the application language provides direct support for concurrency, the implementor should consider mapping **activities** onto the appropriate language feature. Since the Mascot definition demands that the total network of **activities**, **IDAs** and **servers** should be invariant at run-time and, in particular, forbids the dynamic creation of **activities**, language features which allow the creation of additional threads of execution should be made unavailable to the application programmer. The implementor must document how the language facilities have been used to support the Mascot model of concurrency and which facilities have been suppressed.

Any language which directly supports concurrency will probably also provide suitable mechanisms to allow safe and sustained communication between the separate threads of execution. The implementor should consider the language facilities in this area, in the light of the Mascot model, to determine whether or not they are adequate to support inter-**activity** communication through an **IDA** or **server** as prescribed by the Mascot definition.

If the implementor of a development environment elects to utilise the language features for concurrency and inter-**activity** communication, the mapping of the Mascot **activities**, **IDAs** and **servers** onto these language facilities must be described. The mechanisms whereby the **IDA** or **server** designer may ensure the necessary mutual exclusion and cross-stimulation between separate processing threads must also be documented. Furthermore the use of these mechanisms should be restricted to the **access procedures** of **IDAs** and **servers**.

The Mascot Synchronisation Model

Synchronisation in Mascot covers the mutual exclusion of competing processes and the cross-stimulation of co-operating processes. Explicit synchronisation is achieved by four primitives which operate on special objects called **control queues**. Since synchronisation takes place only in respect of access to **IDAs** and **servers**, each **control queue** is conceptually part of the structure of an **IDA** or a

server. Synchronisation primitive operations are only available within an **access procedure**. The **control queue** is defined as an object on which the primitives have the effects defined below, and which may be given a priority specification to influence the scheduling algorithms.

Mutual Exclusion - JOIN and LEAVE

Mutual exclusion of competing **activities** is effected by the calls of the paired primitives JOIN and LEAVE, each of which takes a **control queue** as its single parameter. Between the times when it performs a JOIN operation and the subsequent LEAVE operation, an **activity** is said to be in the queue . A **control queue** is said to be ***empty*** if there is no **activity** in that **queue**.

JOIN and LEAVE form brackets around critical sections of code in which only one of the competing **activities** in the **queue** is allowed to proceed. This **activity** is said to be at the head of the queue and is therefore its owner. The use of the primitive operation LEAVE by an **activity** which is not the owner of the specified **queue** constitutes a run-time error as does the performance of a JOIN operation by an **activity** which is already the owner of the nominated queue. An implementation of a Mascot development environment defines the action that results. It is recommended that the occurrence of an error be recorded and that the **activity** be prevented from further execution.

The algorithm that determines how an **activity** reaches the head of the queue is not defined by the Definition but first-in-first-out is normal practice. The algorithm must be documented.

Cross-stimulation - STIM, WAIT and WAITFOR

A stimulus/response mechanism between **activities** is provided by the primitives STIM and WAIT, each of which takes a **control queue** as its single parameter. An alternative form of WAIT providing a time-out facility is WAITFOR which takes two parameters, a **control queue** and a time delay. The STIM operation may be performed at any time by an **activity**, but the WAIT and WAITFOR operations may only be performed by the **activity** which is the owner of the named **queue**.

An **activity** which performs a WAIT operation on a **control queue** is prevented from continuing its processing unless, or until, a corresponding STIM has been applied to the same **control queue**.

A WAIT operation has one of two possible effects:

(a) If a STIM is being held (see below), that STIM is consumed and the **activity** is allowed to continue.

(b) If no STIM is being held, the **activity** is prevented from continuing until the next STIM is applied.

A STIM operation has one of three effects:

(a) If there is an **activity** WAITing, the STIM is used immediately to make the WAITing **activity** eligible for scheduling.

(b) If there is no **activity** WAITing, and no unused STIM for the **queue** is being held, then the STIM is held for use by the next WAIT operation on the **queue**.

(c) If there is no **activity** WAITing and an unused STIM for the **queue** is being held, then the further STIM has no effect.

The use of the WAIT primitive by an **activity** which is not the owner of the specified **queue** constitutes a run-time error. An implementation of a Mascot development environment defines the action that results. It is recommended that the occurrence of the error be recorded and that the **activity** be prevented from further execution.

The WAITFOR operation is similar to WAIT except that the **activity** which performs it becomes eligible for scheduling on expiry of the time delay, given as a parameter, if no STIM has been applied to the **queue** in the mean time. The reason for the **activity** being released from WAITing may be indicated through a function value return mechanism or output parameter.

Control Queue Ownership - CHECK

An **access procedure** normally contains paired JOIN and LEAVE instructions. This is known as a closed access protocol and it forces correct usage. Alternatively an open protocol may be used in which the JOIN and LEAVE instructions are contained in different **access procedures**. Such an approach allows more discretion to the application program. This can lead to greater simplicity and an improvement in efficiency.

The purpose of the CHECK primitive is to allow an **access procedure** which does not contain a JOIN-LEAVE pair to check that the calling **activity** is the owner of the nominated queue. The primitive takes a **control queue** as its single parameter. If the **activity** which issues the CHECK instruction is the owner of the specified **control queue**, the CHECK primitive returns control to the **activity** with no further action. The use of the CHECK primitive by an **activity** which is not the owner of the specified **queue** constitutes a run-time error. An implementation of a Mascot development environment defines the action that results. It is recommended that the occurrence of the error be recorded and that the **activity** be prevented from further execution.

4.3 DEVICE HANDLING

Introduction

The mechanisms by which peripheral devices are controlled vary significantly from computer to computer. On some machines the contents of device registers are manipulated through normal operations on specific memory locations. Other machines provide special instructions for device control. These instructions may be part of the normal set available to all programs or may be privileged instructions requiring a special mode of compilation for their generation from a high level language. Thus, **modules** which interact with peripherals may require special privileges or facilities not appropriate to other **modules**.

In Mascot, as we have seen in an earlier section, the handling of peripherals is the domain of the **server**. This is the only **class** of **template** allowed to contain code which directly manipulates peripheral control registers or which utilises special facilities provided by the **context** for that purpose. All the necessary facilities may therefore be made available automatically, to the **modules** which require them, during the construction of a **network**.

One of the most important considerations, in connection with the interaction of a real-time systems with a device, is whether the device must be polled in order to detect the completion of an operation, or whether completion is signalled by an interrupt. It is the latter of these two options which is the more common and which requires the specialised facilities described in this section.

Interrupts

An interrupt is a signal, generated by a device, to inform the processor to which the device is attached that a peripheral operation has been completed. Its effect, subject to consideration of relative priorities, is to cause the processor, on completing the execution of its current instruction, to abandon temporarily the current process and to commence execution of instructions stored at a predefined address. This new set of instructions is called an interrupt **handler**. Depending on the computer architecture, a **handler** may be unique to a particular interrupt, unique to the priority level of the interrupt, common to all interrupts or common to a defined sub-set of interrupts.

When control is transferred to a **handler**, the contents of at least some of the registers being used by the interrupted process are saved. This is known as context switching and the amount of information automatically saved, for subsequent restoration when the the process is resumed, also varies considerably with the architecture of the machine. It may be as little as the value of the program counter or the context switch may be so complete as to provide the **handler** with its own register set and, where appropriate, virtual memory map.

The precise nature of an interrupt **handler** is thus determined by the number of devices with which it is concerned and the degree of automatic context switching which takes place when one of these interrupts occurs. In order to minimise the impact of interrupt handling on the normal operation of the system, it is customary to confine the operations performed in the **handler** to a minimum. Thus for a simple device the next operation might be initiated from within the **handler**, while a device which requires a large amount of data to be transferred to or from it, between operations, might be restarted from elsewhere.

Interrupt Handling in Mascot

Given the range of computer architectures to be catered for and the need for efficiency in real-time systems, it is not possible to define a single mechanism for interrupt handling which is completely transportable. Mascot therefore provides a set of facilities which allow suitable solutions to be developed to meet a range of circumstances. These facilities are:

(a) (HANDLER) the ability to declare **handlers** in **server tempates**,

(b) (CONNECT) the ability to associate a **handler** with a particular interrupt,

(c) (DISCONNECT) the ability to disassociate a **handler** from an interrupt,

(d) (ENDHANDLER) the ability to signal completion of processing of the current interrupt and

(e) (STIMINT) the ability to STIM a **control queue** from a **handler**.

Here (a) is a keyword in the design representation language, used in place of PROCEDURE, and the other four facilities are kernel primitives. CONNECT takes two parameters:

```
CONNECT(handler, interrupt_no)
```

to identify the interrupt handler and the interrupt with which it is to be associated, respectively. DISCONNECT is applied to the interrupt identification:

```
DISCONNECT(interrupt_no)
```

STIMINT, which is applied to a **control queue**, differs from STIM (see Section 4.2) only in guaranteeing that it will not result in an immediate reschedule. The documentation of the run-time system in a Mascot

development environment must describe the means by which the above facilities are made available.

Ideally it should be possible to design an interrupt **handler** for minimum execution time and store useage without needing to take any account of the **activity** which has been interrupted. However, this can only be achieved where the hardware provides automatic vectoring to a **handler**, uniquely associated with the interrupt being serviced, and full automatic context switching. Most hardware used for real-time systems is not as sophisticated as this.

In practice, it may be necessary to provide some support in the Mascot **context** even for interrupts which are CONNECTed to **handlers** located in the **servers** of the application software. The following table, which differentiates between different degrees of hardware vectoring to a particular **handler** and different degrees of completeness in automatic register saving and restoring, indicates the range of options.

Vectoring	Register Save	Data Transfer
Hardware	Hardware	**Handler** in **Server**
Hardware	Hardware	Access Mechanism in **Server**
Hardware	**Server**	**Handler** in **Server**
Hardware	**Server**	Access Mechanism in **Server**
Hardware	**Context**	**Handler** in **Context**
Hardware	**Context**	**Handler** in **Server**
Hardware	**Context**	Access Mechanism in **Server**
Context	**Context**	**Handler** in **Server**
Context	**Context**	Access Mechanism in **Server**
Context	**Server**	**Handler** in **Server**
Context	**Server**	Access Mechanism in **Server**
Context	**Context**	**Handler** in **Context**

In the above table **'Server'** is used to denote a **server** in the application software. The term **'Context'** should be taken to include the possibility of a **server** in the **context**.

Design Considerations

There are a number of considerations, some of them imposed by the computer architecture, which influence the design of interrupt **handlers** and their execution. Three of the most significant issues are discussed below.

Nesting of Interrupts

Devices may be classified on a scale of priorities such that a **handler**, in course of execution, may itself be interrupted by a device possessing a higher priority. When the higher priority interrupt has been

handled, control returns to the **handler** of the lower priority interrupt before eventually being returned to the originally interrupted **activity**. Several levels of priority may be catered for in this way. In such circumstances it is acceptable for the individual **handlers** to be somewhat more complex since their execution does not lock out higher priority interrupts. Control is still taken away from the software scheduler, however, and more stack space is required to provide local storage for the **handlers** than in the non-nested case.

Stack Management

Local working space for a **handler** may be provided from the stack of the interrupted **activity**. This avoids the necessity of switching stacks when control passes to or from a **handler**. However, it also means that every **activity** stack must be large enough to accommodate the maximum possible depth of interrupt nesting if this is employed. Alternatively, a separate stack may be allocated either to each **handler** or to each level of interrupt, again allowing for interrupts to be nested, or a single stack can be shared by all **handlers** if there is no nesting.

Pre-emption

After an interrupt has been handled (ENDHANDLER), control may be returned to the interrupted **activity** or, alternatively, the interrupted **activity** may be transferred to the current lists and control passed to the Mascot scheduler (see Section 4.4).The latter strategy results in the better system response at the possible cost of a slight reduction in throughput. If it is adopted, a further question arises as to whether the interrupted **activity** should be added to the head or the tail of the current list. If it is placed at the tail, this results in a 'round-robin' scheduling operation within a priority level (there will normally be a current list for each level). Placing it at the head of the list may be regarded as more 'fair' since then each **activity** is allowed to complete its slice of execution before any other **activity** of the same priority is scheduled. Notice that this is not the same as the co-operative mode of scheduling as higher priority **activities** are allowed to run at any point.

4.4 SCHEDULING AND PRIORITIES

Introduction

In any concurrent system where the number of parallel processes exceeds the number of processors, a scheduling function is required to allocate processing time amongst the competing processes. Each processor, periodically, must cease the execution of one process and commence execution of another. Associated with these re-schedule points there are two decisions which have to be made. The first is to determine when the current slice of execution on a given processor is to end and the second is to select which process is to be allocated the next slice. It is normal practice to assign a priority level to each process to assist in determining, for the purposes of the second of these decisions, which of a number of waiting processes is the most urgent. The choice of scheduling strategy is usually governed by the desire to optimise the response to external events for a given amount of processing power.

In these respects Mascot systems are the same as any other concurrent system. The Mascot definition does not prescribe the use of any particular scheduling strategy though it does require that the selected strategy be documented for the information of users of the development environment. The choice of scheduling algorithm is deliberately left to the implementor in order to allow the optimum algorithm for the application to be adopted. The more important of the possible scheduling and priority schemes are discussed below in Mascot terms.

Co-Operative and Pre-Emptive Scheduling

One of the major factors affecting the responsiveness of a concurrent system is the basic mode of scheduling adopted: co-operative or pre-emptive. Under a co-operative regime reponsibility for the first of the two scheduling decisions, slice termination, is vested in the application rather than in the Mascot kernel. An **activity** continues its slice of execution until it volunteers to give up the processor by invoking one of the kernel primitives. This may be one of the scheduling primitives such as JOIN (on a JOINed **control queue**) or WAIT/WAITFOR (on an unSTIMmed **control queue**) or a re-schedule may be invited more directly by means of the SUSPEND primitive.

Provision of SUSPEND is mandatory in a development environment which supports co-operative scheduling. Its use guarantees re-scheduling provided that there is at least one **activity**, of at least equal priority, waiting to run. If there are no such schedulable **activities**, the **activity** issuing the SUSPEND is re-entered for another slice of execution.

Development environments which support the timing group (see Appendix E) of primitives provide another means of directly surrendering the use of a processor. This is the DELAY primitive which takes a time period, in implementation defined units, as its argument. An **activity** invoking DELAY is not

considered for re-scheduling until the nominated period has elapsed irrespective of whether there are other schedulable **activities** or not. The other primitive in the timing group is TIMENOW which is a function returning current system time The units and range of system time are implementation defined..

There are two other synchronising primitives whose use may, even under co-operative scheduling, result in a re-schedule. These are are LEAVE and STIM when they have the effect of releasing an **activity** which has previously been held up. The decision on whether to re-schedule in these circumstances is implementation defined.

Under a co-operative regime it is guaranteed that, immediately after an interrupt has been handled, a reschedule will not take place. Control is always returned to the interrupted **activity**. This mode of working tends to minimise the time spent in performing context switches; an important consideration on hardware in which the context switch is inefficient. It also leads to simplification of the kernel code and, in the application, exclusive use of a resource can be guaranteed without recourse to special synchronising primitives such as JOIN and LEAVE. There are, however, complicating effects. Contrary to the normal Mascot philosophy, the **activity** programmer must take account of the execution time of any long sequences of code and add calls on SUSPEND at strategic points in order to avoid taking excessively long slices. This is generally unsatisfactory and, indeed, the overheads imposed by frequent calls on SUSPEND may cancel the original gains. In the limit, a faulty **activity** containing a closed loop would bring the remainder of the system to a halt unless the kernel clock handler checks for processor hogging and takes action to TERMINATE the offending **activity**.

Co-operative working effectively limits the ability of the scheduler to optimise the system's response. The alternative is pre-emptive scheduling under which the kernel is free to re-schedule following an interrupt. The arrival of an interrupt is an event which is likely to alter the state of the system. An **activity** which has been awaiting this interrupt may well have become the most urgent. Under pre-emptive scheduling, it can be entered for execution without delay. An **activity** which is hogging a processor is less damaging than under a co-operative regime though it may still be detected and TERMINATEd. The opportunity of re-scheduling after a clock interrupt makes time slicing possible by setting a maximum slice time.

Priority Schemes

The simplest priority scheme, if it can properly be so called, is one in which all **activities** are given the same priority. A single current list, containing all the currently schedulable **activities**, is maintained on a first-in first-out basis. The scheduler always selects the **activity** which has been waiting longest, that is, the **activity** at the head of the list. This scheme is the fastest possible in execution but provides little scope for improving the response of the system to external events. Indeed this can only be achieved by the crude mechanism of adding calls on SUSPEND in order to increase the number of re-schedule points. As the length of the current list cannot be determined by an **activity**, such a strategy is not very effective and may, in addition, prove wasteful of processor resources if carried to excess.

In the next priority scheme to be considered, several priority levels are employed to which all **activities** are allocated, once for all, on first being executed. A first-in first-out list is maintained for each priority level and these are scanned by the scheduler in priority order to select the **activity** at the head of the first occupied list. The choice thus falls on the most urgent task which has been waiting the longest. Compared with the single priority approach, this strategy results in greater flexibilty at the expense of greater complexity and, in general, a heavier drain on processor time. It possesses the advantage that the response to external events can be tailored by changing the relative priority levels of **activities** without altering the coding of any **templates**. The response problem is not, however, totally solved since no matter how high its priority level an **activity** may be held up in the pending list of a **control queue** behind one of lower priority.

Finally, in an attempt to overcome this problem of the collision of two **activities**, of different priorities, competing for ownership of a **control queue**, multi-level priority schemes may be extended to encompass dynamically varing priorities. Cross-stimulation between co-operating **activities** is not affected. Although, at first sight, variable priorities may seem to present an added complexity, they can be easier to use in practice than fixed priorities. This is because a priority level can be associated with a resource in such a way as to reflect its scarcity and this is often easier to assess than the average urgency of executing a particular **activity**. Two of the many possible algorithms which may be used to determine priorities in this type of scheme are discussed below.

A static priority may be associated with a **control queue**. Any **activity** which JOINs such a **queue** assumes this priority if it is higher than its own current priority. The **activity**'s priority reverts to its previous value when the **control queue** is released. This scheme is relatively easy to implement although the restoration of priority on LEAVE can present some difficulties if multiple JOINs and LEAVEs are not nested. It does, however, complicate the coding of the primitives and extend the execution time required by JOIN and LEAVE. Also the lower priority **activities** execute at higher priority than is desirable when there is no collision.

As a further refinement, **control queue** priority may itself be made variable. Initially minimum priority is allocated and the priority of an **activity** JOINing the **queue** is not affected. However when an **activity** is placed on the pending list, the priority of the **control queue** is set to the higher of its own and that of the **activity**. Thus the priority of the **queue** owner is maintained as that of the highest priority **activity** in the **queue**. The **queue** reverts to minimum priority when the pending list becomes empty. The result is that **activities** are executed at their assigned priorities except when in collision with a higher priority **activity**. This improvement is achieved at the cost of further complication of the code and increased execution time for JOIN and LEAVE.

4.5 MULTI-PROCESSOR CONFIGURATIONS

Introduction

Multi-processor target configurations for Mascot applications can be classified in a variety of ways on the basis of their distinguishing characteristics. They may differ, for example, in respect of any of the following:

(1) Inter-Processor Relationships. There are three principal types of configuration. One processor may be designated as master with all others acting as slaves. A symmetrical arrangement of identical processing elements is possible in which the storage and input/output resources of the system are all shared. Finally the processors may operate autonomously each with its own private set of resources.

(2) Memory Visibility. The possibilities here are that the system's memory resources may all be accessible to all processors, may be divided into units each of which is available for use by one processor only or a combination of these arrangements may be employed with a mixture of common and private storage.

(3) Number of Mascot Kernels. Multi-processor configurations may contain a single copy of the Mascot kernel or many copies, for example one per processor.

(4) Allocation of **Activities**. The selection of a processor for the task of executing a Mascot **activity** may be performed on either a static or a dynamic basis.

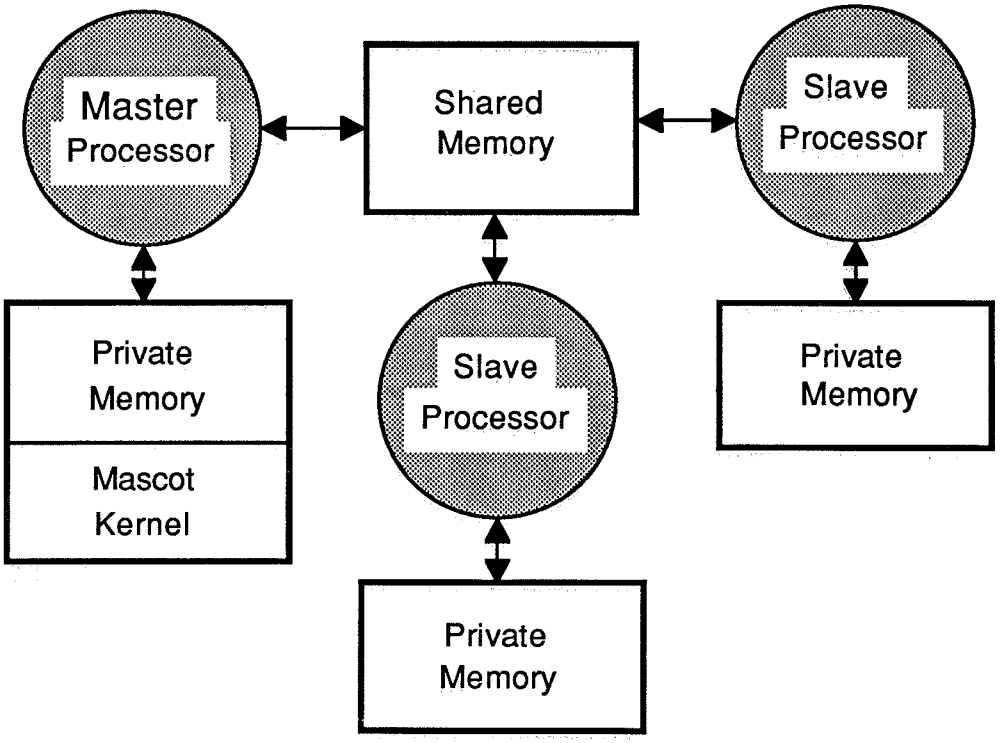
It is also possible to envisage complex configurations employing, within the same system, more than one of the possibilities listed under (1) and (4) above. Two classes of configuration using at least some shared memory are discussed below; those configurations employing a single Mascot kernel being considered first and then those with multiple kernels. Finally configurations with no shared memory are discussed very briefly.

The following discussion is in terms of a **control queue** based implementation. Similar considerations apply to any alternative run-time implementation strategy and, when considering such alternatives, the arguments should be interpreted accordingly.

Single Kernel Configurations

In a master/slave arrangement, the single kernel has one processor, the master, dedicated to its execution and so need not be re-entrant. The diagram below, which illustrates this configuration, shows

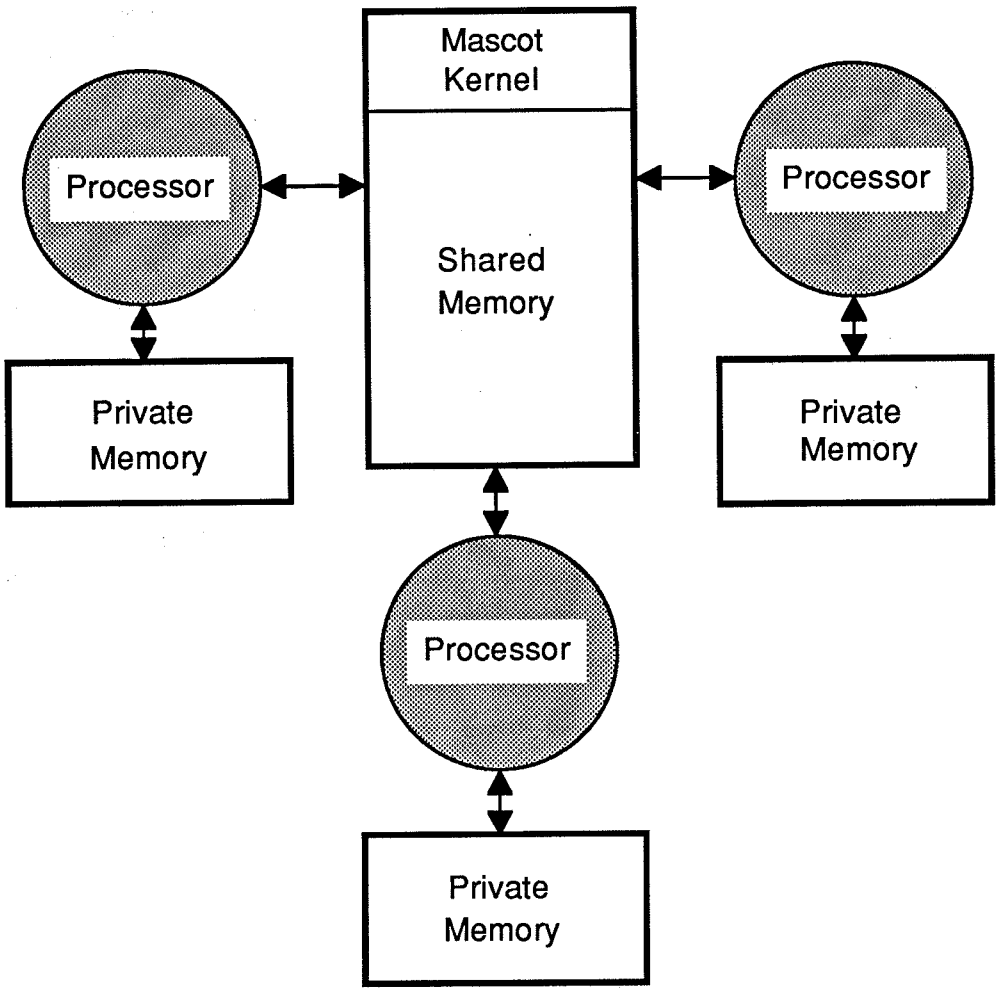
two slave processors although there might be any number. The simplest example of this arrangement, where there is just one slave processor, comes nearest to the familiar single processor case and, in general, the software configuration is simple.



As the kernel data are accessible to only one processor no protection is required. Equally, an **activity** being executed on a slave processor has no direct access to kernel facilities and some software mechanism must be provided to permit such **activities** to invoke primitive operations. Some common memory is therefore essential.

In a symmetrical arrangement, the processing elements are identical and almost all memory and input/output facilities must be common. A three processor, symmetrical configuration is illustrated in the diagram below. The common store contains the system's single, multi-thread kernel. Separate copies of read-only components may be held in local memory for efficient access while there is potential lockout on shared kernel data and the application's control queues must be protected for concurrent access. Control of the kernel passes from one processor to another, but only one can be 'master' at any one time. An **activity** could be executed on different processors for different slices.

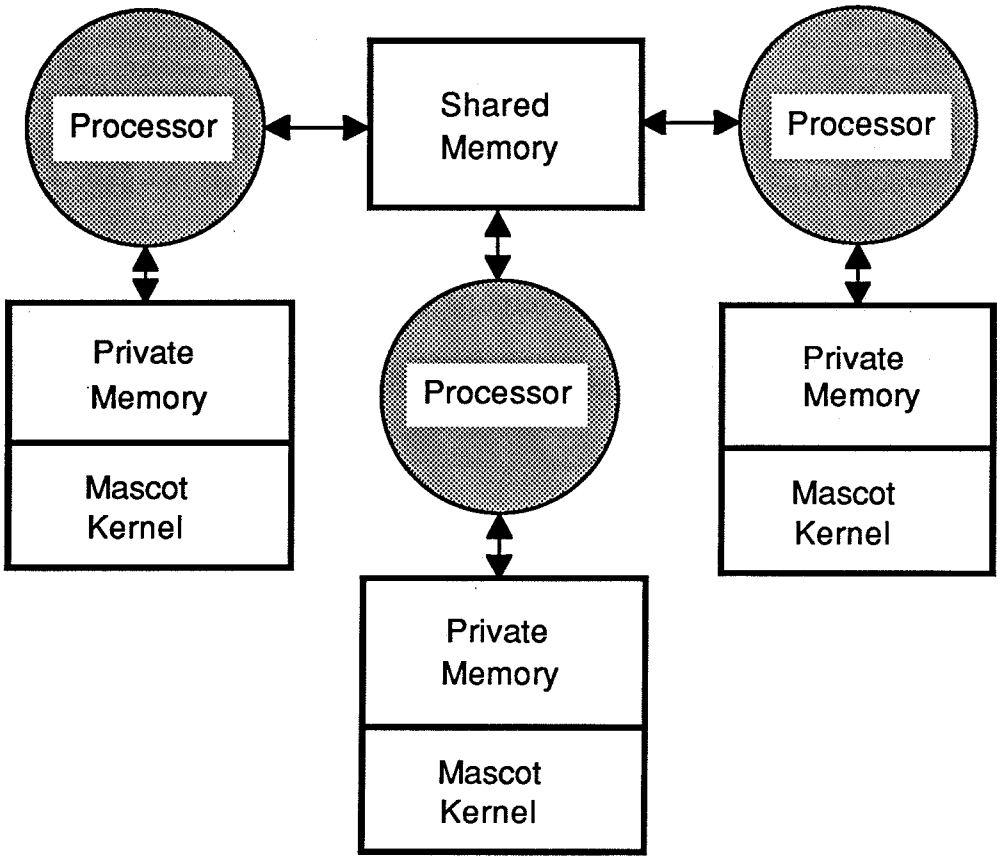
A major advantage of this type of configuration is that the workload of the processors is balanced. If one processor fails, the system is able to continue with reduced performance or with support for some of its functions withdrawn. Extra processors can be added with minimal **rebuilding**. This is an attractive arrangement provided sufficient common memory can be made available without significant access time penalty.



Multiple Kernel Configurations

Providing each processor with its own copy of the Mascot kernel, in private memory, leads to the autonomous arrangement (illustrated below) in which the processors act independently and the kernels perform almost as in a single processor configuration. Each processor has private input/output devices but some shared memory is desirable. **Activities** are assigned, permanently, to a particular processor at **build**-time. Kernel data, such as for example pending lists, may need to be shared between kernels.

The key consideration which further distinguishes configurations of this type is the means by which application or **context** software running on one processor can communicate with software running on another processor. At the most fundamental level, communication between processors can be by polling or can be interrupt driven. At a level more relevant to the present discusson the choice is between communication effected directly by the application software or by the kernel acting on its behalf. The extent of the common memory available is one important factor in determining the method.



Communication between Mascot **activities** involves **access procedures**, **control queues** and **IDA** data. With total support from the kernel, and given an adequate amount of shared memory, communication among **activities** assigned to different processors may take place in a wholly transparent way. The sharing of **IDA** data presents no problem. The sharing of **access procedures** avoids duplication of code at the cost of the additional access time associated with common memory. It is the sharing of **control queues** which requires special support from the kernel.

When all primitives are available for application to shared **control queues**, the pending list of such a queue may contain **activities** whose execution is administered by different kernels. Ownership of the queue passes from one kernel to another. A kernel needs to be aware, therefore, when one of the **activities** under its control has been STIMmed remotely (that is, by an **activity** running on another processor). This information can be supplied by polling or by interrupt.

The main advantage of such an arrangement is that a component of the application software can be relocated for execution by a different processor without any changes being necessary to its source **template**. On the other hand, the need to hold pending lists in shared memory and the continual change of processor owning a shared **control queue** is not very desirable for efficient and fault-tolerant operation.

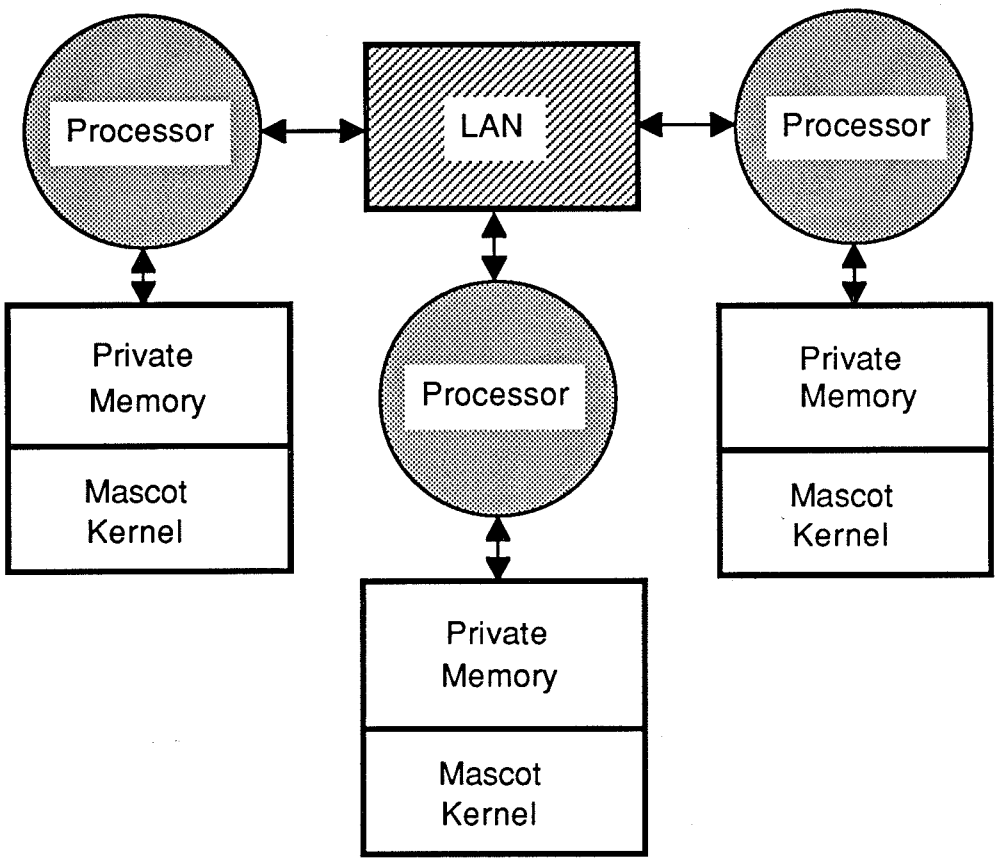
An alternative approach is one in which inter-processor communication is restricted to the ability of an **access procedure**, running on one processor, to apply a STIM to a **control queue** controlled by the

kernel of another processor. The pending list for such a **queue** only contains **activities** which belong to the local kernel so that ownership of the **queue** remains with a single kernel. Thus while **IDA** data are shared, pending lists can be held in local memory. The operation of a remote STIM involves drawing the attention of the local kernel to the fact that a WAITing **activity**, under its control, is now available for scheduling or, if there is no **activity** WAITing on the **queue** and no previous STIM being held on that **queue**, that there is now a STIM to be held for future use. All that need be communicated, in order to convey this information, is the identity of the **control queue** involved. This can be done through shared memory, allowing all the **control queues** to be held in local memory.

This method reduces the overheads of a multi-processor target to a minimum. However, communications restrictions between processors prevent the 'processor boundaries' from being transparent to the design and/or implementation of the application. Thus, changes in the distribution of components may require changes to the network.

Communication Linked Configurations

Finally it should be mentioned that multi-processor working may take a form in which there is no common memory. Communication, in this case, is effected by a 'communications bearer', such as a local area network (LAN), independently of the kernel. Some support may, however, be offered by the **context**.



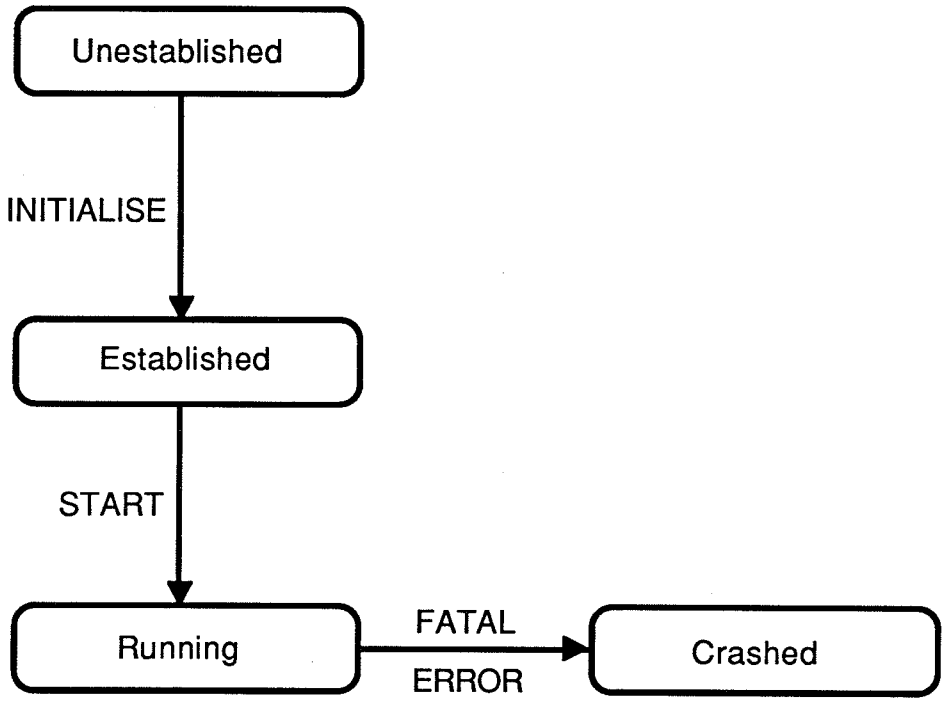
4.6 EXECUTION CONTROL

Introduction

In this section the execution control facilities which are mandatory in a Mascot development environment are presented first. There follows a discussion of the additional facilities which it is desirable should be provided to support software commissioning. A set of commands is then described for the application of control functions to individual **activities**, **IDAs** and **servers**. Finally further additional facilities are described which support the hierarchic control of **networks**.

Mandatory Facilities

When a Mascot **system** has been **built**, each of its components is said to be in an **unestablished** state. Before any constituent **activity** can be executed it is necessary that it, and the **IDAs** and **servers** to which it is connected, be initialised. Part of the process of establishing an **IDA** or **server** is the execution of any initialisation procedure contained. When a component has been successfully initialised, it achieves the state value of **established.** An **established activity** may be started once all the **IDAs** and **servers** to which it is connected have been **established**. Thus, the minimum necessary set of execution control facilities consists of INITIALISE which establishes a component and START which sets an established **activity** running. All Mascot development environments must, therefore, support these two functions. The corresponding state change diagram is given below.

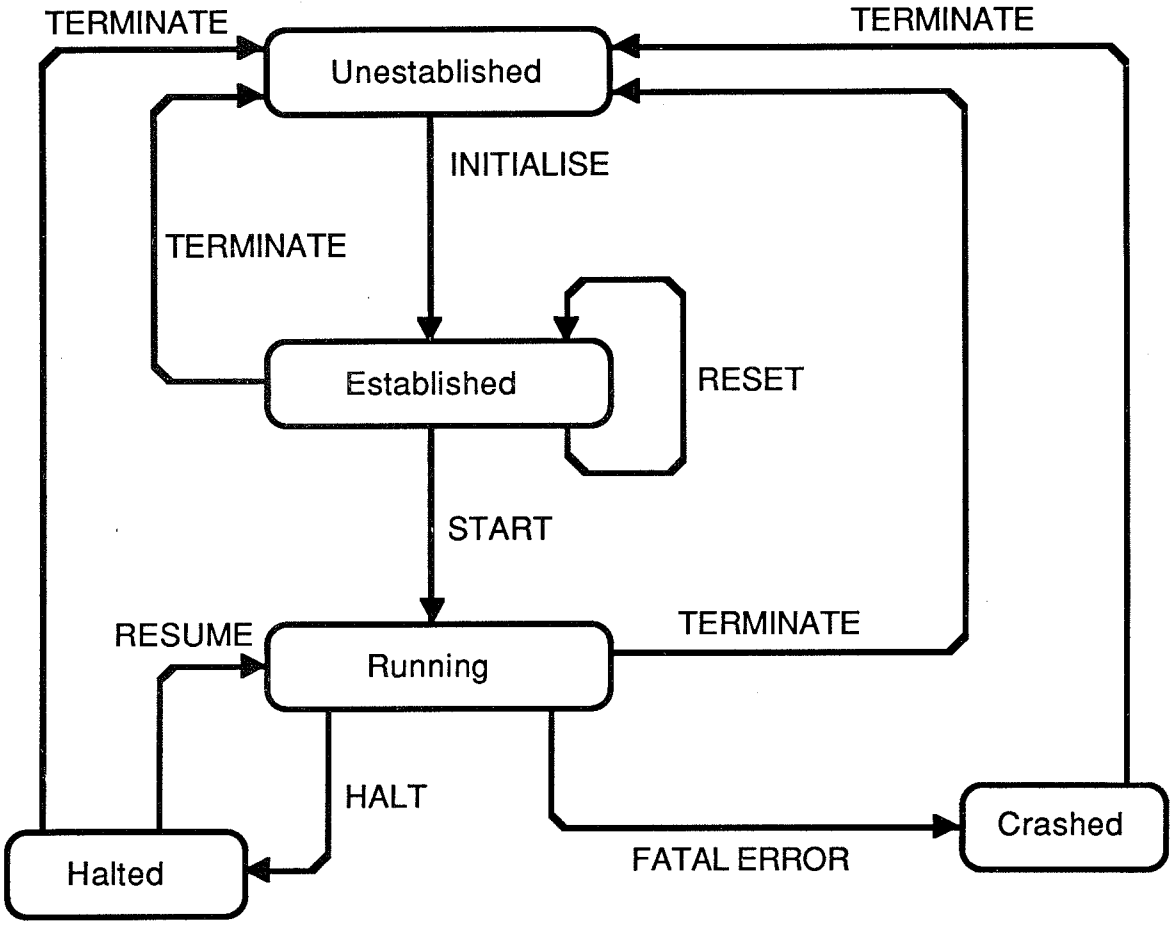


An **activity** enters the **crashed** state after a fatal run-time error has been detected.

The two mandatory functions may be provided either as part of the **build**-time facilities or as on-line, run-time commands. The mechanism of the former approach is implementation defined. In the latter case, unless the mechanism is automatic, the functions should be made available through an appropriately named interface which may be defined either as part of the **context** or as an application **module**. Any parameters will be expressed in a form which is consistent with the other language facilities. A development environment may also provide a command interpreter which uses the interface.

Additional Non-Mandatory Facilities

While the mandatory facilities described above fulfil the minimum requirements for an operational Mascot **system**, additional facilities are desirable during the software commissioning phase of a project. A Mascot development environment may therefore provide four further functions, HALT, RESUME, RESET and TERMINATE which should be made available as on-line, run-time facilities. Associated with these functions is a further state: **halted**. The full state change diagram is shown below.



The **halted** state is entered as a consequence of the application of the HALT function. While in the **halted** state, an **activity** is not eligible for scheduling although a timeout may expire or the **activity** may become the owner of a control queue.

The full Mascot definition allows **activities**, as well as **IDAs** and **servers**, to contain initialisation procedures (see Appendix E). Termination procedures may also be included in all three of these **module classes** and reset procedures may be included in **IDAs** and **servers**. Where such procedures are catered for in a Mascot development system, their execution is initiated as part of the INITIALISE, RESET and TERMINATE functions. Note that once INITIALISE has been applied it cannot be re-applied without first returning to the **unestablished** state by means of TERMINATE. The reset procedures of **IDAs** and **servers** may, however, be executed in the **established** state by means of the RESET function.

The functions HALT, RESUME and TERMINATE are not normally considered to be suitable for use in an operational system because of their drastic side effects. In the case of HALT, an **activity** is prevented from further execution until RESUMEd. If the **activity** is executing an access mechanism of an **IDA** or **server**, this may prevent other **activities** completing execution of access mechanisms in the same **IDA** or **server**, and hence will potentially stop the remainder of the **system**. The operation of TERMINATE also has drastic side effects in that an **activity** may be aborted while executing an access maechanism. This, in general, can result in the contents of **IDAs** or **servers** becoming inconsistent. While these side effects are tolerable during software commissioning, they are not normally tolerable during system operation. If there is no alternative to the use of HALT/TERMINATE, it may be necessary to introduce special facilities to prevent the side effects.

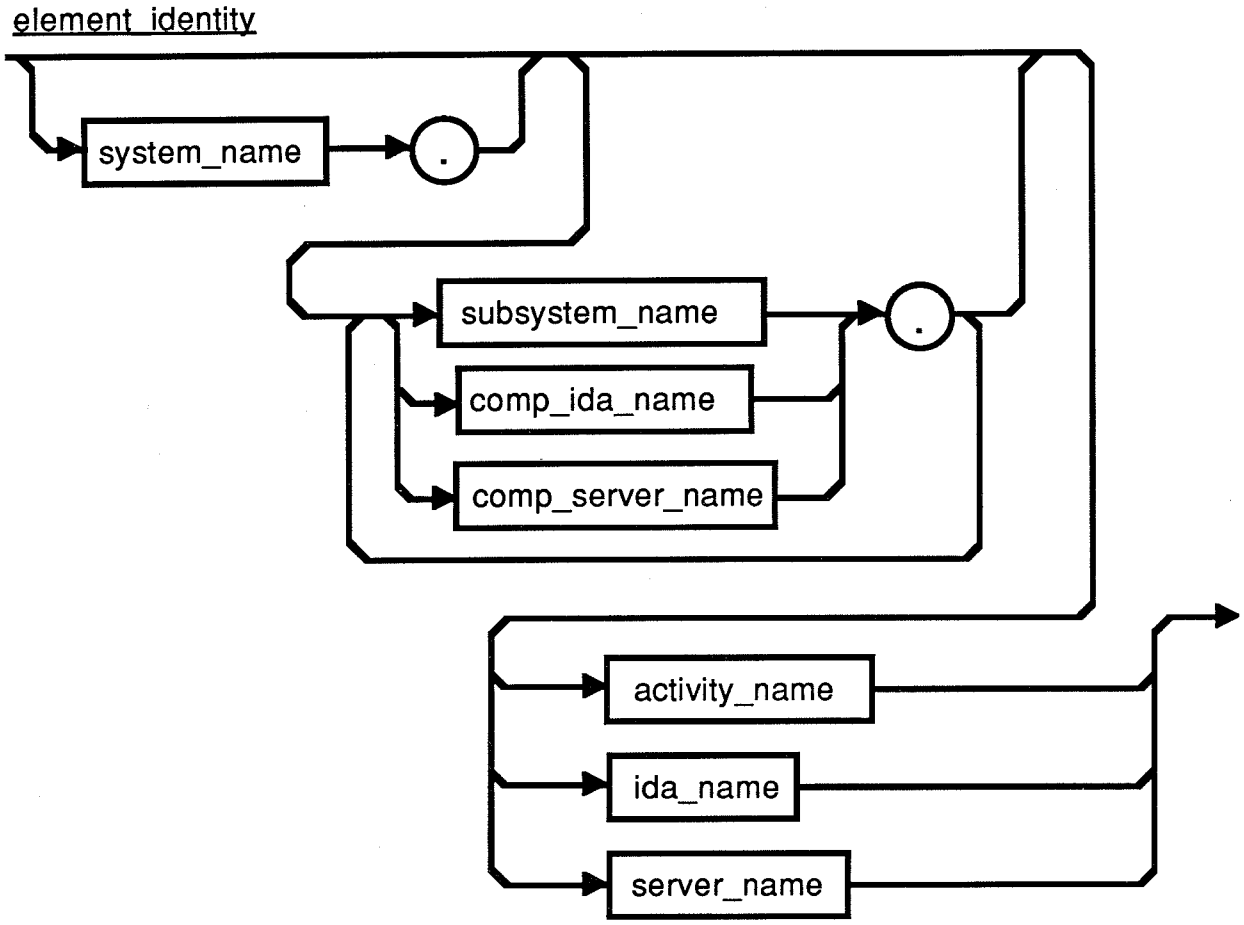
Command Description

The INITIALISE Function

If we assume all its **components** to be in the **unestablished** state, a complete **system** may be initialised in the following manner. First the INITIALISE function is applied to each **IDA** and **server**. If the operations are successful, all these **components** will become **established**. It is then possible to **establish** the **activities** by applying the INITIALISE function to each of them in turn. This ordering may be relaxed where it is guaranteed that the initialisation code within an **activity** does not interact with any of the **IDAs** or **servers** to which it is connected.
The effect of applying the INITIALISATION function is to cause the initialisation procedure to be executed. After successful application of the INITIALISATION function an **activity**, **IDA** or **server** is in the **established** state.

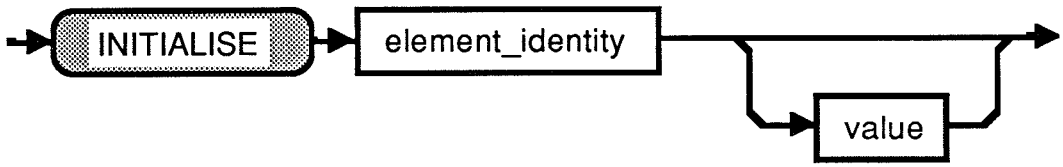
A run-time system may place restrictions on the facilities available to an initialisation procedure provided that these restrictions are documented. In particular, since the initialisation procedure may be invoked by the run-time system, rather than by an **activity**, it may be inappropriate to use the **control queue** primitives other than STIM or STIMINT.

In a development environment which supports the INITIALISE function on-line there must be at least one parameter, namely the **activity**, **IDA** or **server** identity whose preferred form is defined by the syntax diagram below:



The **system** name may be omitted in circumstances where only one **system** is allowed so that there is no possible ambiguity.

If the passing of a parameter to **IDA** or **server** initialisation procedures is supported, then means of specifiying the value to be passed will be defined. Where a command interpreter is provided the command will take the form shown below.

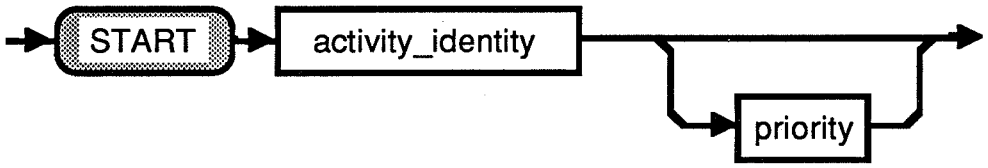


The START Function

The START function can only be applied to an **activity** which is in the **established** state, and for which all the connected **IDAs** and **servers** are also **established**. The effect of its successful application is to put the **activity** into the **running** state and to inform the scheduler that the **activity** is eligible for

scheduling from its initial entry point.

In a development environment which supports the START function on-line, the function accepts either one or two parameters. These are the element identity and, if the development environment supports priority change at start, a priority value. Where a command interpreter is provided, the command will take the form shown below.



Where a priority value is accepted it overrides the the priority specified during **building**. If dynamic priority adjustment is supported, this value may itself subsequently be overridden. If the START function cannot be applied to the **activity** then an error or warning message will be generated to specify the identity of the **activity** and the reason for the failure. For example:

START: Error: Activity STABILISE.FILTER is crashed
START: Warning: Activity MMI. COMMANDINT already running

The criterion which determines the nature of the message is whether the desired effect has been achieved or not. Thus attempting to START a **running activity** should generate a warning while attempting to START a **crashed activity** should result in an error message.

The HALT Function

The HALT function may only be applied to an **activity** which is in the **running** state. Its effect is to put the **activity** into a **halted** state and to inform the scheduler that the **activity** is not eligible for scheduling. Where a command interpreter is provided the command takes the form:

The function accepts a single parameter which is the identity of the **activity** to be **halted**. If the function cannot be applied to an **activity** then a warning is generated to specify the **activity** identity and the reason for the failure. For example:

HALT: Warning: Activity STABILISE.FILTER already halted

The RESUME Function

The RESUME function may only be applied to an **activity** which is in the **halted** state. Its effect is to put the **activity** into the **running** state and to inform the scheduler that the **activity** is now eligible for

scheduling. Where a command interpreter is provided the command takes the form:

The function accepts a single parameter which is the identity of the **activity** to be resumed. If the RESUME function cannot be applied to the **activity** then an error or warning message will be generated to specify the **activity** identity and the reason for the failure. For example:

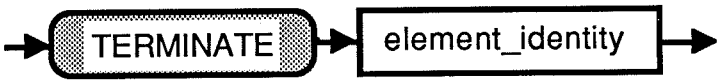
```
RESUME: Error: Activity STABILISE.FILTER is crashed
RESUME: Warning: Activity MMI.COMMANDINT already running
```

The TERMINATE Function

The TERMINATE function can be applied to an **activity** which is in any state but **unestablished**. If **IDA** or **server** termination procedures are supported then TERMINATE may be applied to an **established IDA** or **server** provided that all connected **activities** are either **unestablished** or **established.**

The effect of the TERMINATE function applied to an **activity** is to put it into the **unestablished** state and inform the scheduler that the **activity** is not eligible for scheduling. The **activity** is removed from all internal queues and lists within the run-time executive and is removed from all control queues it has joined.

Where a command interpreter is provided the command takes the form:



The function accepts a single parameter which is the identity of the **activity**, **IDA** or **server** to be **terminated**.

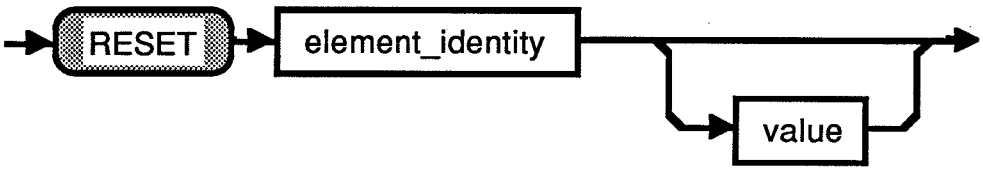
The effect of the TERMINATE function applied to an **IDA** or **server** is to execute the termination procedure if such is supported by the development environment.

If the TERMINATE function cannot be applied to an **activity**, **IDA** or **server** then an error or warning message will be generated to specify the identity and the reason for the failure. For example:

```
TERMINATE: Error: Channel STABILISE.MESSCHAN is connected to running
                  activities
TERMINATE: Warning: Activity STABILISE.SHIP is unestablished
```

The RESET Function

The RESET function may only be applied to **IDAs** and **servers** which are in the **established** state. Its effect is to cause the execution of the reset procedure and it leaves the element **established**. Where a command interpreter is provided the command takes the form:



The second parameter, if provided, is passed to the reset procedure. If the RESET function cannot be applied to an **IDA** or **server** then an error message will be generated to specify the identity and the reason for the failure. For example:

RESET: Error: IDA STABILISE.MESSCHAN is unestablished

The RESET function provides a means of re-initialising an **IDA** or **server** without needing to TERMINATE the associated **activities**.

Hierarchic Control Facilities

The control facilites described above operate on individual **elements**. Mascot development environments may be extended to support hierarchic control of **systems**, **subsystems** and **composite IDAs** and **servers**. These facilities operate by applying the functions already described, recursively, to the **elements** and lower level **networks** within a specified **network** entity. Provision of these features necessitates some extensions to the function descriptions and these are discussed below.

A development environment is required to define the effect of an hierarchical control command failing to operate. A much wider range of exceptional situations can occur such as, for example, the application of the RESUME function to a **network** which contains a **crashed activity**. It is worth noting that individual **components** of a **network** can be in different states making the concept of an overall **network** state meaningless.

Hierarchic Identity

The preferred form of identity described above for **activities**, **IDAs** and **servers**, must be altered to allow the omission of trailing fields. The function is then applied to the named structure. In order to limit the possibilities for error, actions applying to the **system** as a whole should require the system name to be specified. It could however still be omitted for **subsystems** and **composite IDAs** and **servers.**

INITIALISE Applied Hierarchically

The hierarchic form of the INITIALISE function should first initialise all the **IDA**s and **servers** and then the **activities** of the **network** being initialised. If the development environment is not capable of determining the necessary order of initialisation according to the dependencies then it must provide a mechanism for specifying this.

The development environment must define the method of handling parameters of initialisation procedures.

START Applied Hierarchically

The development environment must define how the start priority value is to be interpreted in connection with a **system** or **subsystem**.

HALT, RESUME and TERMINATE Applied Hierarchically

The order of application must be defined for the development environment. If these functions are provided by means of an interface which is accessible to the application then the development environment must ensure correct operation when a command is issued by an **activity** which is itself within the scope of that command.

RESET Applied Herarchically

The hierarchic form of RESET is similar to the hierarchic form of INITIALISE except that it only affects **IDA**s and **servers**. Again, the development environment must define the method of handling parameters.

4.7 ERROR HANDLING

Introduction

As stated in the Introduction to this Handbook, Mascot is aimed primarily at the development of real-time, embedded software. Such applications usually demand a high degree of fault tolerance. In the face of a wide range of run-time errors, systems are expected to continue operation, albeit in a degraded mode in which some functions are no longer available or are available at reduced efficiency. Error detection and handling are therefore of considerable importance in the operational software. During the development stage, it is essential to have good facilities for error reporting and for some applications this facility must be retained, possibly in a modified form, when the system becomes operational.

The three functions: detection, handling and reporting involve actions at three levels of the system: the hardware, the executive software (the Mascot kernel) and the application software.

Mascot Error Handling Requirements

Errors may be detected at all three levels and handled by either of the software levels. At the hardware level, such events as arithmetic overflow or underflow and memory protection violation are detected and normally signalled by means of an interrupt mechanism. Response to these signals may be provided by either the executive or the application software. In the case of the application level, the coding which checks the hardware status for errors may be programmed explicitly or may be included implicitly by the compilation system.

Other errors are detected directly by the software. The executive detects such faults as the illegal use of primitives and, under a co-operative scheduling regime, an excessive time slice. In the application software, checks may be made explicitly for errors related to the logical significance of the program. An example might be a mutually inconsistent set of data values. For a wide range of errors, the division between those which must be catered for explicitly and those which are automatically trapped by code embedded by the compiler, depends heavily on which implementation language is being used and its provisions for data typing. Languages such as Pascal and Ada provide the programmer with a great deal more assistance in these matters than, for example, Coral 66. Range checks on individual values fall into this category.

Turning now to the question of handling the error once it has been detected, this, in fault tolerant systems, is largely the responsibility of the application software. Consequently, error handling in the executive, whether of error signals originating from the hardware or of directly detected errors, normally consists of passing the information on to the application. In a few instances, however, the executive may take the necessary action itself in a manner which is transparent to the application. In a paging

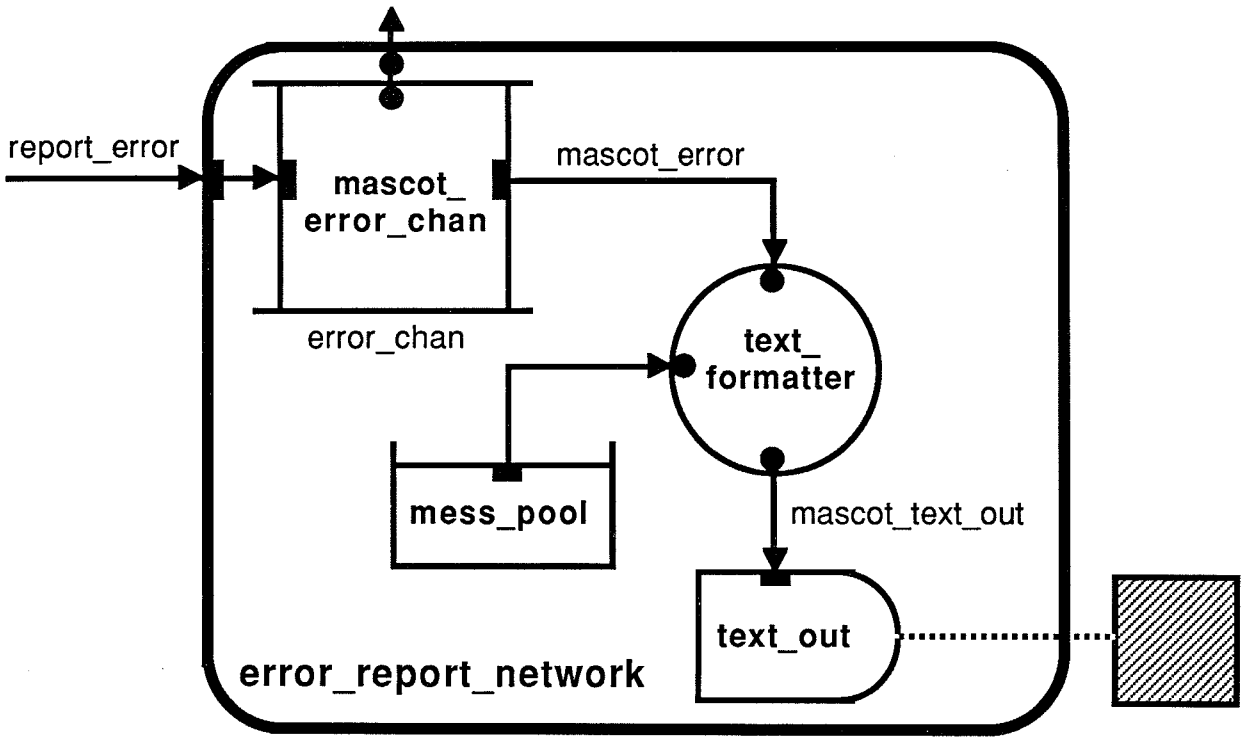
implementation, a page fault could be dealt with in this way. In other cases, where the error is too serious for recovery, the execution thread containing the error may be aborted and a message transmitted to the error reporting system. Where an error has been detected by or communicated to the application software, the mechanism whereby control is passed to the appropriate handling code is again very language dependent with Ada, PL/1 and RTL/2 providing built-in facilities of varying degrees of sophistication.

The Mascot definition calls, under various circumstances (see for example Section 4.2), for the generation of error or warning messages. Concern with the foregoing is otherwise limited to the possibility that analysis of the message may be used to trigger recovery action. A Mascot development environment is required to provide error reporting facilities which may be used by both the executive and application parts of the software in a uniform manner. These facilities are defined in terms of three elements: an **error notification interface**, an **error channel** and a standard **error reporting network**. Support for these elements is mandatory only during development and not in an operational Mascot system.

Example Error Handling Facility

In this section error reporting facilities are described, for convenience, in the form of a particular implementation. This implementation is not part of the standard. The documentation of a Mascot development environment must specify the precise mechanisms which are supported and suitable means of amending these facilities should be provided.

The diagram below illustrates how the standard facilities could be provided.



The purpose of the **network** is to take messages, convert them to a suitable textual form and transmit them to an appropriate device. Messages passed to the **network** are stored initially in a **channel** of type ***mascot_error_chan*** . Input is via a **window** which provides the standard **error notification interface** which, in our example, has been named ***report_error*** .

```
ACCESS INTERFACE report_error;
   PROCEDURE error(message : text_string; .....);
   PROCEDURE fatal_error(message : text_string; .....);
END.
```

Thus there are to be two **access procedures** ***error*** and ***fatal_error*** each of which place in the **channel** a message which contains, in addition to the text itself (possibly represented by an error number), an indication of the severity of the fault and the identity of the **activity** which originated the message (or on whose behalf it was originated by the executive). Other, user generated, components may also be included in the message. The procedures may take other parameters as defined in the documentation of the development environment.

Procedure ***fatal_error*** has the additional effect of setting the state of the **activity**, in which the fault has occurred, to **crashed** and informing the scheduler that the **activity** is no longer eligible for scheduling. In the above example **network** these actions are represented by a **port** which propagates information out of the **error channel**. If the **activity** is selected for monitoring, calling either **access procedure** will result in the generation of an appropriate monitor record (see Section 4.8).

The **error channel** provides an output **window** of type ***mascot_error*** to give access to the messages which have been generated by the system. The **access interface** takes the form:

```
ACCESS INTERFACE mascot_error;
   WITH error_message;
   PROCEDURE get_error(VAR message : error_message; .....);
END.
```

where

```
DEFINITION error_message;
   TYPE
      severity = (error, fatal);
      error_message =   RECORD
                           text : text_string;
                           fault_type : severity;
                           act : act_id;
                           .
                        END
   END.
```

Any additional parameters of **access procedure** ***get_error*** are defined for the Mascot implementation.

The development environment documentation will also define other characteristics of the **error channel** including its behaviour when full, the queuing algorithms (eg priority levels of messages) and the behaviour where no consumer **activity** is connected to the **channel**.

The standard **error reporting network** is required to contain facilities for taking messages from the **error channel**, via the ***mascot_error*** **window**, converting them to textual form and transmitting them to the output device via a **window** of type ***mascot_text_out*** . In our sample system these facilities are illustrated as an **activity** ***text_formatter*** with access to a message pool and connected to a **server window** of the appropriate type. **Access interface** ***mascot_text_out*** is of the form:

```
ACCESS INTERFACE mascot_text_out;
   PROCEDURE text_put(mess : text_string);
END;
```

The format of the message displayed by the device is as follows:

activity identity : error type error message parameter(s)

where the preferred form of an activity identity is

subsystem name . activity name

The 'error type' takes its value from the ***severity*** component of the message (error/fatal) and any additional parameters must be defined for the implementation.

It should be possible for the user to adapt the standard error reporting facilities by replacing the **server** and/or replacing the processing element. In the latter case the application error handling might analyse the error message and determine some network level recovery action. For example, receipt of a warning message that a channel has reached 95% of capacity might trigger an action to filter incoming data more heavily in the hardware and so reduce the number of events being processed.

4.8 MONITORING

Introduction

The problems of testing and diagnosis in a real-time, multi-threaded system differ considerably from those encountered during the corresponding phase in the development of a sequential program. In particular, the standard interactive tools will usually be inappropriate in that it is neither possible nor desirable to suspend execution of the system or to execute it in single step mode. The Mascot approach encourages the planning of testing and diagnosis during the design phase of a project and a Mascot run-time system should provide a range of facilities to assist in this aspect of the design.

The use of the set of facilities provided for this purpose is known as monitoring and the primary purpose is to collect and display an ordered list of significant interactions within a system being commissioned. These interactions may be between the elements of the system or between these elements and the **context**. It is obviously desirable that the collection of this data should, as far as possible, be a function of the run-time system rather than of the application software. The monitoring facility should provide, therefore, a means of recording the interactions without the need to incorporate special code in the application **templates**.

The basic concept employed in Mascot is that the collection of monitoring information should be decoupled from the processing and display of that information. The recommended collection system maintains the time ordering of the information gathered and is designed to have a minimal impact on the normal execution of the components under investigation. The remaining facilities include provision for selecting subsets of the possible information and the means of processing the gathered data and displaying it in a readable form.

Monitoring is presented, in this section, in the form recommended for a run-time system which supports the Mascot primitives (see Section 4.2). For a run-time system which does not support these primitives, some form of monitoring facility should still be provided. In this latter case, the range of events to be monitored is more difficult to define but is likely to be broadly similar.

Events to be Monitored

In a run-time system which supports the Mascot primitives, the events which would be candidates for monitoring include the following:

Control Queue Primitives:

JOIN

WAIT

WAITFOR

LEAVE

STIM

STIMINT

CHECK

Occurrence of Errors:

ERROR

FATAL_ERROR

Timing Primitives:

DELAY

TIMENOW

Interrupt Related Primitives:

CONNECT

DISCONNECT

ENDHANDLER

Interrupt occurrence

Scheduling Operations:

Selection of an **activity** for scheduling (**START_SLICE**)

Ending of period of **activity** execution (**END_SLICE**)

Tracing of Control Flow:

Automatic recording (where provided for by the compiler) of such events as procedure entry and return (**TRACE**)

Miscellaneous Primitives:

SUSPEND

ENDROOT

The monitoring facility also provides a mechanism for recording, in addition to the events listed above, application specific events (record points). This mechanism is known as the RECORD facility and is normally provided in the form of one or more primitives which have implementation defined parameters.

Selection

In even the smallest application, the volume of data gathered and displayed by the monitoring facility is potentially overwhelming. It is therefore necessary to be able to select sub-sets from the totality of monitorable events. The control mechanisms, recommended for this purpose, provide a set of filters which select the desired sub-set dynamically. Experience has shown that the most useful criteria are:

Which **activities** are to be monitored?

Which primitives are to be monitored?

Which **control queues** (where relevant) are to be monitored?

Which record points are to be active?

Is the automatic trace data (if any) to be recorded?

For an event to be recorded, all of the relevant selections must be in force. Thus, for a **control queue** primitive, the **control queue**, the primitive and the **activity** must all be selected.

The recommended control facilities are provided by the operations SELECT and EXCLUDE which respectively extend and reduce the set of events to be recorded. These operations require two parameters. The first of these is the identity of the group to which it applies:

ACTIVITY
PRIMITIVE
CONTROL_Q
RECORD
TRACE

and the second is the identity of the member of that group. The recommended form of reference to **activities**, **control queues** and record points is by the element identity as defined in Section 4.6 of the Handbook extended by the addition of the structure:

.controlq_name

to the identity of the **IDA** or **server** containing the **control queue**, and by the addition of the structure:

.record_point_name

to the identity of the element containing the record point.

The hierarchic form of element identity, described in the Section 4.6, may be used as a shorthand means of SELECTing or EXCLUDing a collection of **activities**, **control queues** or record points in a single operation. Thus, by the omission of trailing fields, reference can be made, for example, to all the **activities** in a **subsystem**, all the **control queues** in an **IDA** or all the record points in an **activity**.

Identification of the TRACE records to be monitored is entirely dependent on the identification information which the development system associates with each trace point.

Primitives should be selected by the names used under the heading of 'Events to be Monitored'. The term SLICES should be used to represent the START_SLICE, END_SLICE pair, and ERRORS to represent ERROR and FATAL_ERROR.

Group references may be supported in a Mascot monitoring facilty through the keywords:

ALLPRIM	- All primitives
ALLACT	- All **activities**
ALLCQ	- All **control queues**
ALL RECORD	- All record points

All selections remain in force until explicitly excluded by means of the EXCLUDE function.

It will be appreciated that there are some elements of any system which must not be selected for monitoring. The operations of the monitoring facility itself are an example. The system builder should therefore provide facilities to inhibit monitoring of nominated **activities** and **control queues**. These inhibitions should be transmitted to the run-time monitoring system and should be used to override the effects of individual selection of the specified items or the effects of ALLCQ and ALLACT.

In summary, the parameters of the SELECT and EXCLUDE functions are:

PRIMITIVE	primitive name
ACTIVITY	activity name
RECORD	record point name
TRACE	(additional parameters implementation dependent)
ALLACT	
ALLCQ	
ALLPRIM	
ALLRECORD	
SLICES	
ERRORS	

Recording

It is recommended that the events selected for recording by a Mascot monitoring system are written into the monitor buffer, in strict order of occurrence, in one of the following modes:

REAL_TIME
IN_LINE

In the default mode of IN_LINE, all the selected events are captured. If necessary execution of the **activities** being monitored is held up in order to ensure that no monitoring information is lost. In this mode of operation it is permissible to disallow selection of interrupt and scheduling events.

In REAL_TIME mode snapshots of the monitored events are taken while the system is allowed to run with the minimum of interference. Data are written to the monitor buffer until it becomes full. When this has happened, all further monitor data are discarded until the contents of the buffer have been processed for display. The number of monitored events lost in this way should be reported to the processing function.

Three further options are recommended for the control of recording:

HOLD
EMPTY
OFF

HOLD prevents further entries being written to the monitor buffer while preserving the current contents. Further data are discarded until REAL_TIME or IN_LINE mode is selected.

EMPTY clears the buffer and causes further data to be discarded until REAL_TIME or IN_LINE mode is selected.

OFF disables all monitoring (both recording and processing) until REAL_TIME or IN_LINE mode is selected.

These five options may be selected using the SELECT operation with the appropriate parameter.

Processing

The function of processing is to convert the coded information in the monitor buffer into a readable form and display it by means of a suitable peripheral. Facilities must be provided to perform this either on-line or off-line.

The on-line facility allows processing to be turned either on or off. It is controlled by the SELECT and EXCLUDE operations, in the normal way, using the keyword PROCESS. While processing is disabled, it is recommended that the contents of the monitor buffer be circularly overwritten in either REAL_TIME or IN_LINE mode of operation. This ensures that the buffer always contains a record of the most recent events ready for processing and display as an aid to problem diagnosis.

The off-line facility should permit the contents of the monitor buffer to be examined following a system crash.

CHAPTER 5

THE MASCOT METHOD

5.1 METHOD AND USE

Introduction

This section describes the method whereby a design may be derived through the use of the Mascot design representation language and graphical notation set out in earlier sections of the Handbook. The domain of applicability of Mascot includes the development and subsequent maintenance of software for large, distributed, embedded, real-time data processing systems. Without excluding the possibility of applying sub-sets of the notation and the method to smaller systems, Mascot places particular emphasis on the word 'large'. Large, that is, in the sense of large numbers of people involved in the development, a large amount of program text to be written, a large number of requirements to be serviced simultaneously, a great variety and quantity of hardware resources, and a project whose development extends over a long time scale. The adoption of this emphasis on large systems has resulted in the evolution, in Mascot, of the techniques necessary for handling the scale and complexity which seem to be inescapable features of modern software developments.

The Mascot design method provides a basis for managing both the concurrency in development which arises when many people are deployed simultaneously on a task, and the iteration which arises when lower level design causes previously taken higher level design decisions to be changed. The method can be viewed as a process where there are multiple sites of execution and where earlier stages may need to be revisited.

Development Management

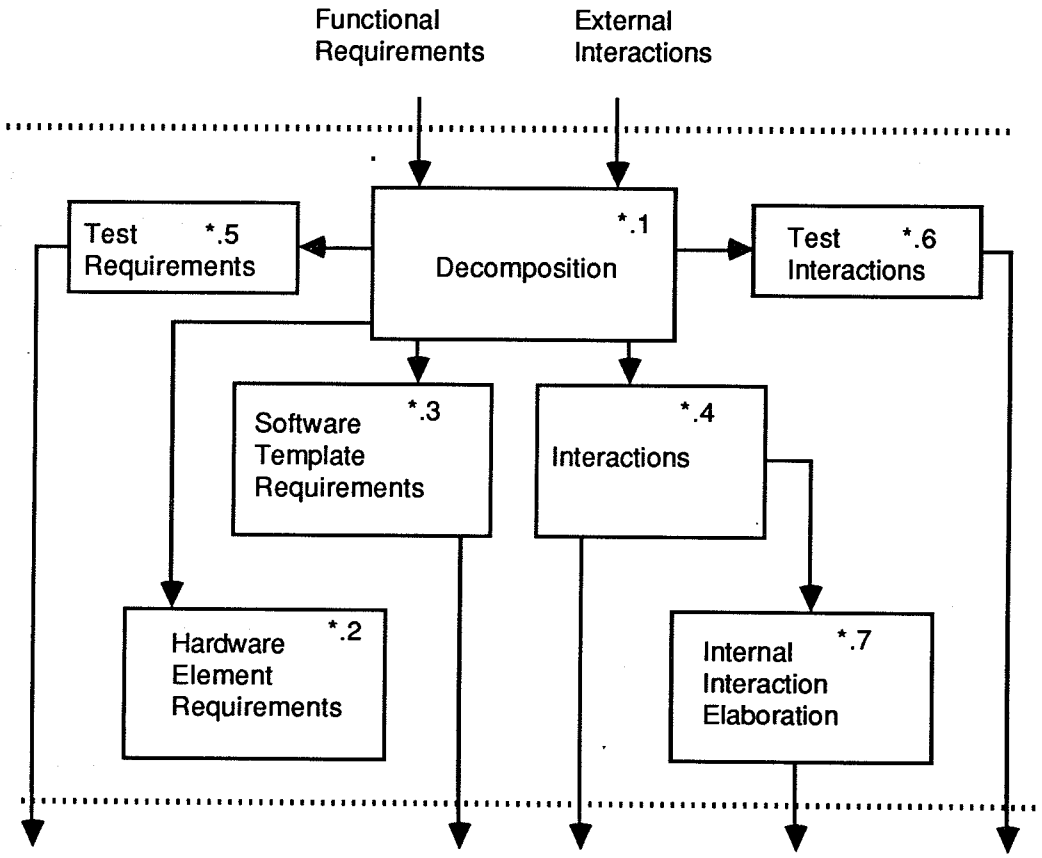
The Mascot method is based on the progressive and repeated application of a simple but powerful technique: that of alternating the requirement and design viewpoints in the course of establishing an hierarchic structure. At each level of decomposition, a design is postulated to meet a set of requirements. Implementability is thus kept firmly in mind when elaborating the design structure. This encourages a closed loop approach to the development process whereby the functions provided and the performance achieved by a design at any level can be related, directly and easily, to a corresponding set of requirements.

Great emphasis is placed on design visibility throughout software development. This is achieved by using the various **composite** structures to give an hierarchic form of design definition. In this way large problems can be partitioned to contain the complexity at any point in the design, and to allow work to be shared amongst members of a team. Skill, experience and good management are required during this process which is essentially creative, but is highly visible and amenable to control.

Mascot is not prescriptive; thus it does not provide any recipes for establishing or filling out the framework for a given design problem. However, it does permit integration with a wide range of complementary software system analysis and design methods. In this way the benefits of Mascot structural description and development control can be combined with the advantages provided by other well proven or emerging software engineering techniques.

Design Decomposition

Each software component of a Mascot system is derived from a **template** and for every **template** there is a uniquely defined design task. The Mascot method, therefore, includes stages which relate to specific classes of **templates**: the operational **system**, **networks**, **elements** or **subelements**, **simple templates** and test **systems**. Each of these stages is further elaborated in terms of substages. The diagram below, in which each box represents a substage, summarises the common approach used in all stages.



General Decomposition Diagram

At any point in the decomposition process a design is postulated which will satisfy the external interactions and functional requirements derived from earlier stages. Each external interaction is an identifiable set of operations and/or data flow constraints and each functional requirement specifies a transformation which relates to operational and/or data flow effects on one or more interactions. Design

decomposition at any stage identifies a set of **components**, each with its own functional requirements and interactions. For each software **component**, the requirements for the corresponding **template** are then derived (substage 3) together with its interactions (substage 4). These, in turn, lead to the functional requirements and external interactions for further decomposition. Note that the Mascot graphical conventions give explicit identification of interactions: a dashed line indicates interaction between a **component** and a device, a **path** indicates an interaction within a **network** and a **link** indicates an interaction within an **element**.

A **template** has to be designed to meet functional requirements and to service or generate external interactions which are determined by previous stages of the design. Thus, for example, the design of the operational **system** may place requirements and interaction constraints on a component **subsystem** whose **template** in turn places requirements and interaction constraints on a component **element**. The **element template** may be further decomposed and, in the process, places requirements and interaction constraints on component **subelements**.

In the decomposition of a **template** (substage 1) the following are identified: any hardware elements to be employed, the software components and the internal interactions. The characteristics of the hardware elements are described in substage 2. The **template** specification and functional requirements for each **component** are defined in substage 3. The semantic and dynamic properties of the internal interactions are defined in substage 4 while the syntax of the interfaces is defined in substage 7. Finally, the functional requirements and interactions associated with testing the **template** are defined in substages 5 and 6.

The results, with the exception of those produced by substage 2, flow forward as data for lower levels of decomposition. The complete design process is bounded, at the beginning, by the given framework in which development is to take place and, at the end, by programming the indivisible **elements** and **subelements** which form the atoms of the design, and by designing test **systems** for each of the **templates**.

Technical Authority

From a development management point of view, the responsibilites relating to each design task (each **template**) may be expressed in the roles played by a designer and a technical authority. The designer of a **template** is responsible for carrying out all the substages relevant to the **template** design task. These include the integration of the components identified in the **template** in order to be satisfied that the **template** requirements and interaction constraints have been met. The technical authority for a **template** is responsible for design task definition and for the verification and acceptance of the work carried out by the designer of that **template**.

Normally the technical authority for a **template** will be the designer of the enclosing **template** (in which it is identified). This ensures that the technical authority is responsible for all the requirements and interaction constraints relevant to the design task to be undertaken by the designer; these include:

(a) Interaction Descriptions. These are the definitions of the data flows and operations on the **paths, links** and device interactions which **components**, to be created from the **template**, must generate or respond to (that is the output from substage 4 of a previous design task).

(b) Interface Specifications. These are the formal specifications of the procedures in the Mascot **interface modules** (that is the output of a previous substage 7).

(c) **Template** Functional Requirements. These define the transformations, expressed in terms of the interactions described above, to be carried out by any **component** created from the **template**.

Some formality must be attached to the transfer of work between a technical authority and a designer. Before a design task is placed by a technical authority on a designer, a Requirements Review must be carried out. The formality of this review will be dependent on the extent of the definition detail at the time the design task is initiated. Mascot allows work to proceed against incomplete design definitions (for example, with **specification modules** at **registered status** only) and this partial statement of requirements must be subject to some sort of review. Later, when full **specifications** are available, a more thorough review can be undertaken.

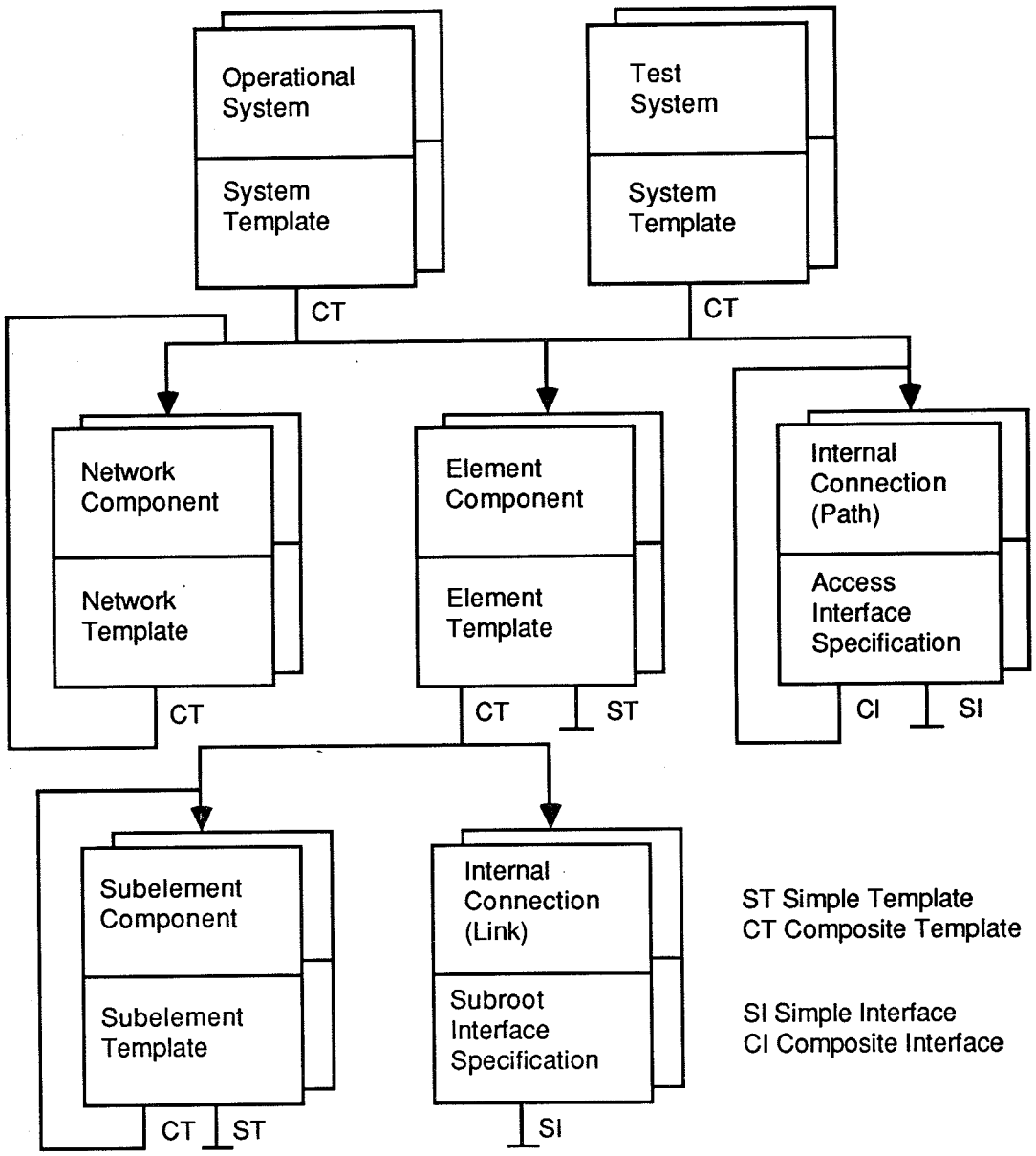
When the design task has been completed a Verification and Acceptance Review must be carried out. This will involve, at the least, a review of all the substages of the design task. Additionally, special Verification Analyses and Acceptance tests may be undertaken at this point.

Design Definition

The principal structural features of Mascot are summarised in the diagram below. This shows decomposition from **systems**, through **networks** and **elements**, down to **subelements**, with **paths** and **links** being identified on the way.

At the top of the structure are the operational **systems** which identify collections of **components** to meet primary operational requirements, and test **systems** which identify a mixture of operational and test **components** for some intermediate test purpose. The diagram shows the closely coupled relationship between **templates** and **components**. During development, the structure of the application software is evolved in terms of a set of interconnected **components** to be created in the execution environment. Each **component** is designed in terms of a **template**. Each connection is designed in terms of an **interface specification**. This approach encourages design abstraction and supports the creation of

multiple **components** derived from the same **template**, or the creation of the same type of **component** in different execution environments for prototyping, testing or re-use.



Design Structure

The principle of 'containment of complexity' should ruthlessly be applied during design structure elaboration. At each stage of decomposition a significant measure of partitioning should be achieved but without generating overly complex internal **component** structures. The final hierarchical design structure should contain the minimum number of levels consistent with the ability to see easily how each **component**, at any level, plays its part in satisfying the requirements generated by the next level up.

Application of the Mascot method is likely to result in a large number of names. These names must be chosen with care and, in the case of very large systems, it may be necessary to introduce special naming conventions and/or measures to limit the potentially global scope of **template** names. The value of

clear, meaningful names cannot be over emphasised. **Template** names should reflect the general functional capability of a **module**, whereas **component** names should be chosen to indicate the particular role of the **component** in the **composite module**.

Not shown in the diagram are various additional **modules** which are not involved in the primary decomposition in terms of **networks**, **elements** and **subelements**. The set of additional **modules** and the purpose for which they are required is as follows:

(a) **Context Interface**. This defines the operating environment of the Mascot software.

(b) **Library Interface**. This defines a set of facilities to be provided by a **library component** within the application software. It may, on occasions, represent the existence of a piece of (possibly custom designed) hardware.

(c) **Library Template**. This defines a set of **library** algorithms.

(d) **Definition**. This defines one or more data types and associated constants.

(e) **Build Modules**. These define the mapping of software design components onto target hardware.

During development, the definition of the design structure will evolve in parallel and be subject to iterative change. This process will be recorded in the **Mascot database** using the **status** progression facilities. The **Mascot database** is similar to a Data Dictionary but has far more emphasis placed on the relationship between entries. The structure at any time captures the current state of the design and allows assessment of the degree of completion.

The quality of the documentation at the end of development will significantly affect the maintainability of the system and the potential of any of its **templates** for re-use. When the development task is 'finished', the documentation set must include the rationale for the 'final' structure. Those parts of the design which have been discarded during the iterative development process need not be retained in full, although the history of the development and any major lessons learnt should be summarised in a design record.

Potential for Re-use.

Mascot is essentially a form of software **template** technology. The output from the decomposition process (substage 1) expresses the design in terms of the functionality of, and interactions between, the **components**. When the interactions are defined (substage 4) their descriptions are generalised so as to maximise the potential for re-use. In the same way, in the definition of the **template** (substage 3), the transformations to be performed by any **component** derived from the **template** are expressed in terms of the generalised interactions derived above. It is possible to identify a number of circumstances frequently encountered when designing a **template** for a particular **component**:

(a) Unique **Template**. This is where a **template** must be designed from scratch. Nothing exists (or there is no knowledge of it) which can be used as a starting point for the **template** design.

(b) Adapted **Template**. This is where a **template** can be designed using an existing **template** as the base line for the development.

(c) Common **Template**. This is where a common requirement exists at several different places in the overall design structure and where it is possible to satisfy this requirement with a common design. Control of the common design should be exercised from at least the lowest common point above the places where it is used in the design hierarchy. A common **template** may be parameterised in various ways, or may be produced in variant forms using automatic program generation techniques, to give some flexibility in use.

(d) Standard **Template**. This is similar to a common **template**, but with applicability across a range of projects.

(e) Existing **Template**. A existing unique or adapted **template** may be adopted for use elsewhere. In being so used it is made into a common or standard **template** and control of its design should be exercised accordingly. Whenever it is decided to make use of an already existing **template** it is likely that this will to some extent affect the interactions and **component** requirements in the enclosing design structure; these must now be tailored to accomodate the existing item.

Design Stages

As already indicated, the design process can be broken down into a number of stages and substages. This must not be construed as a linearisation of the essentially iterative and concurrent design actions which take place in a large software development. However some means of identifying different aspects of the design process is needed and it is for this reason that the stages and substages are defined, each being given a number for ease of reference. Thus, although the work of a development will not proceed strictly in accordance with the stage numbering, the sequence indicates the order in which the design of individual items is elaborated.

The rationale for the 'final' design, or any design which is thought to be good at any particular point of development, may well give the impression that it has been arrived at by orderly sequential application of the design stages. Indeed it is important that it is possible to describe the design in this way. The Mascot method is specifically geared to the problems of controlling the evolution of a design, during which mistakes may be made and requirements changed. The method maximises the formality with which an evolving design structure can be expressed, whilst maintaining flexibility by allowing changes to be made in well defined localised regions.

Before embarking on the first stage of the design it is necessary to begin by establishing the framework in which the development will take place. Although work on this is started right at the beginning of the project it is likely that the bulk of the detail will need to be deferred until the first stage of decomposition has been completed. Subsequent stages may well result in addition, deletion or amendment to the results of this preliminary work. The following aspects need to be considered:

(a) Hardware Environment. This includes establishing the number and type of processors to be used, the size and type of each main storage unit, the means by which hardware modules are to communicate and the number and nature of the peripheral devices which are to be handled by the system. It is necessary to complete this to a degree which is adequate to support any hardware interactions in subsequent stages of the design, and to identify any processing or communication constraints which may affect the design structure.

(b) Development System. This includes the facilities provided by the programming and project support environments together with such items as the means of prototyping and the provision of test harnesses. All these facilities are central to the software development and must be well defined early on.

(c) **Context(s)**. This involves establishing the primitives and standard support facilities which are to be made available through one or more Mascot **context interfaces**.

(d) **Libraries**. These are standard support facilities which are to be created outside the **context** and made available through **library interfaces**. Both **libraries** and **contexts** are directly supported by the Mascot design representation facilities; their definition must be completed in time to support the programming stage of the work.

The method includes six main stages as follows:

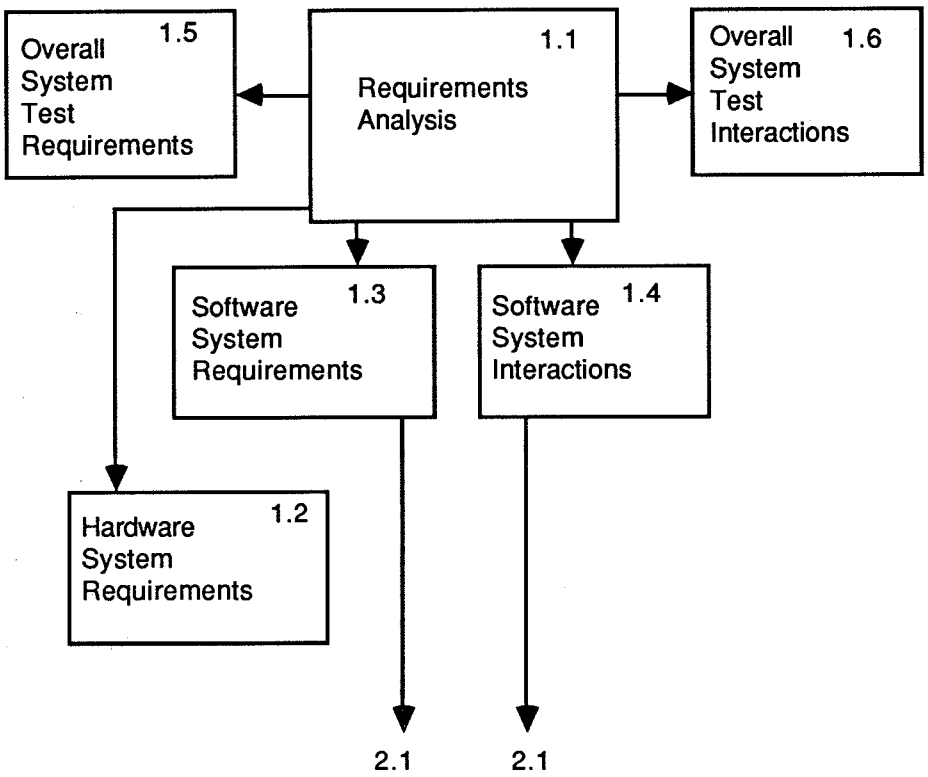
Stage 1 : External Requirements and Constraints
Stage 2 : Design Proposal
Stage 3 : Network Decomposition
Stage 4 : Element Decomposition
Stage 5 : Program Definition
Stage 6 : Test System Definition

Each of these will be illustrated by means of an individual diagram derived from the general one already presented. The seven substages, introduced earlier in connection with the general decomposition model, represent specific design actions related to the decomposition process in each of the six main stages. Work on the various substages of any particular stage will proceed in parallel and need not be complete before work is started on further stages of decomposition. Some substages are not present in all the six stages, as will be illustrated by means of the individual diagrams, and some may optionally be

omitted if they are not required in the elaboration of a particular design item. An arrow leaving a box in any of the diagrams represents a flow of information to a further stage of decomposition (in some cases within the same stage of the method such as where a **network** is decomposed into lower level **networks**).

Stage 1 External Requirements and Constraints

This stage establishes the general requirements and external constraints. The substages of which it is comprised are illustrated on the diagram below and are now described individually.



Stage 1

Stage 1.1 (**Requirements Analysis**) involves analysis of the complete system (hardware and software) with particular emphasis on identification of the software system requirements and interactions. Analysis techniques compatible with the data flow and network principles of Mascot (such as CORE, SSADM, JSD etc.) are particularly relevant during this stage.

Stage 1.2 (**Hardware System Requirements**) describes those parts of the system whose functions are to be performed by hardware units. They are not, in the Mascot method, to be subjected to further decomposition (box 1.2 possesses no output arrow) but such hardware units are not of course precluded from containing independent items of software.

Stage 1.3 (**Software System Requirements**) describes what the software has to do in terms of transformations of data and responses to events originating outside the software system.

Stage 1.4 (Software System Interactions) describes the nature and purpose of each implicit interaction between the software system and the external hardware or software (including communication with an operator).

Stage 1.5 (Overall System Test Requirements) describes the way in which the complete system (hardware and software) is to be tested.

Stage 1.6 (Overall System Test Interactions) identifies particular test data, expected results and the operator control sequences required to carry out the tests on the complete system. No data is carried forward to later stages as such tests are applied to the total system operating in its natural environment.

Stage 1 must be well advanced before any further stages of the development work are started. Although this stage does not produce any formal Mascot design definitions (apart from the name of the system **template**) it is likely to involve the application of formal techniques for requirements analysis. Notice that this stage produces the requirements and device interactions for a Mascot software system which may itself contain embedded hardware components (where these are deemed to be supportive of the software design rather than placing constraints upon it as does the 'non negotiable hardware' discussed before embarking on this stage). There could also be several software systems, rather than just one, interacting through intermediate hardware. The overall system test requirements (stage 1.5) and interactions (stage 1.6) are not, in principle, needed until the end of the development when hardware and software have to be integrated; however stages 1.5 and 1.6 should be carried out as early as possible since they provide a useful measure of requirements analysis verification. Indeed in all stages which generate test definitions, the fact that the corresponding tests are not to be performed until later should not be allowed to delay work on their definitions.

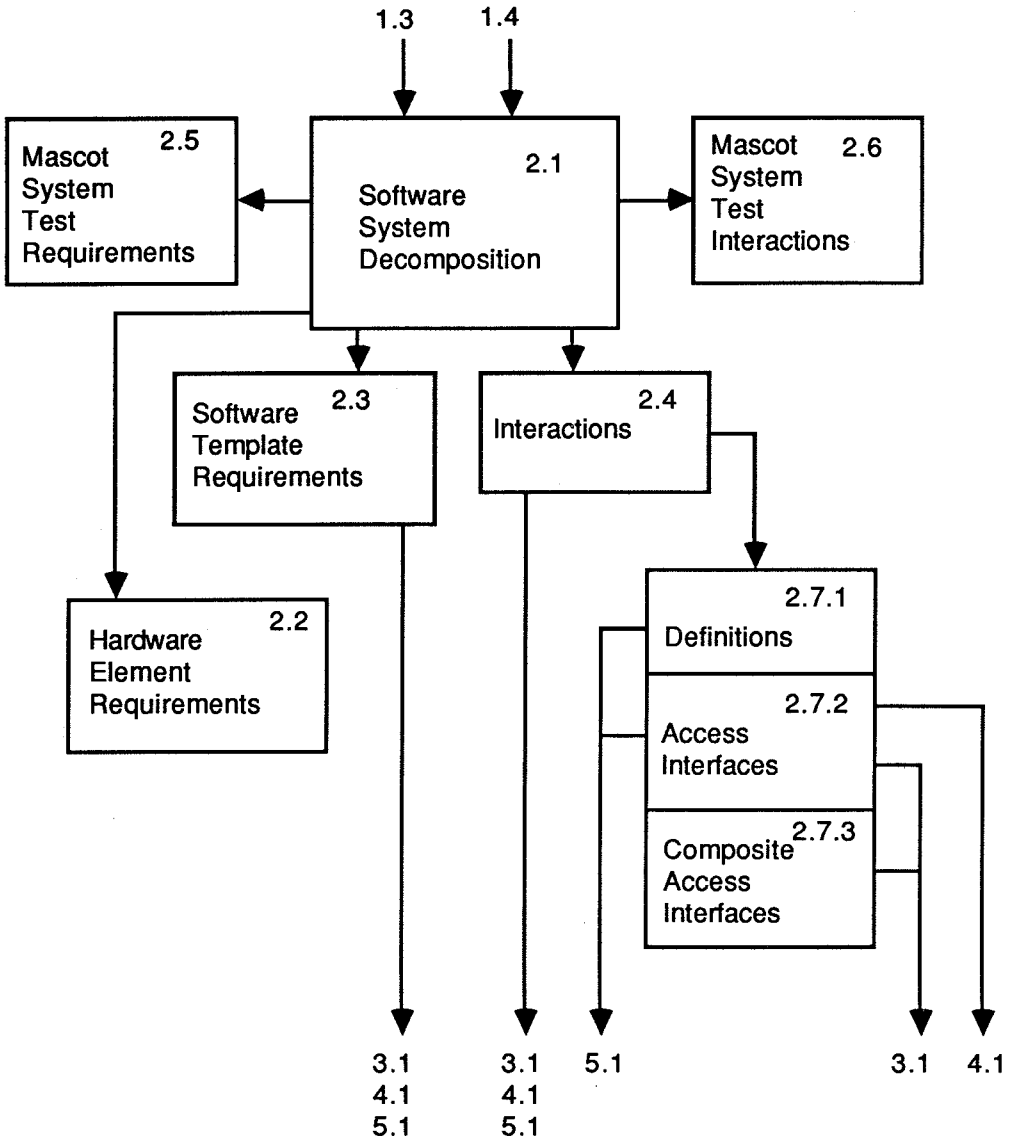
Stage 2 Design Proposal

This stage results in a top level design proposal based on the software system requirements and interactions identified in the stage 1 Requirements Analysis. It is the first point at which a Mascot design structure emerges. The corresponding diagram is shown below.

Stage 2.1 (Software System Decomposition) describes the top level application software design in terms of a Mascot **system**.

Stage 2.2 (Hardware Element Requirements) describes what each of the items of supportive hardware appearing as top level components of the Mascot **system** has to do.

Stage 2.3 (Software Template Requirements) describes the **template** requirements for each software component in the top level software **system** design.



Stage 2

Stage 2.4 (**Interactions**) describes the nature and purpose of the internal interactions (**paths**) between **system components**.

Stage 2.5 (**Mascot System Test Requirements**) describes the way in which the Mascot **system** is to be tested. No data is carried forward to later stages as such tests are applied to the total software system.

Stage 2.6 (**Mascot System Test Interactions**) describes the particular test data, expected results and the operator control sequences required to carry out the Mascot **system** test.

Stage 2.7 (**Internal Interaction Elaboration**) identifies the **definitions** (2.7.1), **access interfaces** (2.7.2) and **composite access interfaces** (2.7.3) required to describe the **paths** between **system components** and the types of the data which flow along them.

No further stages of decomposition can be started until at least the **system template** has achieved **enrolled status**, the **components** have achieved **introduced status**, and the **access interfaces** have achieved **registered status**. In addition, the internal **paths** must be described in terms of semantic and dynamic properties and any serial ordering constraints which apply to data flow along the **paths**. When the **template** requirements have also been defined then this is sufficient to allow further stage 3 or stage 4 decomposition. The stage 5 program design of an **element** or **subelement** connected by an **interface** identified in stage 2 cannot be started until the **definitions** (2.7.1) and **access interfaces** (2.7.2) have been established. Likewise the detail of the **composite access interfaces** (2.7.3) must be complete before a **composite path** can be expanded in a subsequent stage 3 **subsystem** decomposition.

The result of the initial attempt, at this stage, to identify a **component** as a **subsystem** (to 3.1), or an **element** (to 5.1) is necessarily provisional. The ultimate decision as to whether a **template** should be further decomposed can only be made in the light of its designer's more detailed examination. In the case of an **element** this examination may indicate that further decomposition is desirable. In these circumstances stage 5 is abandonned and stage 3 or stage 4 invoked, as appropriate. It is also possible that subsequent decomposition fails because design constraints cannot be met. In this case stage 2 must be repeated.

Stage 2 is the point at which the Mascot method starts to exert its influence in relating software design structure to external requirements and constraints identified during stage 1. Ideally, the top level of software design expression would be expected to contain the following **components**:

(a) A **subsystem** (or **activity**) for each major system function.
(b) A **subsystem** (or **IDA**) for every major system data requirement.
(c) A **subsystem** (or **server**) for every major interaction with external devices or software outside the system.
(d) A **subsystem** for every systemwide internal communication requirement.

For a large application such an ideal would be impractical, resulting in an unmanageable number of top level **components**. However, these **components** would be expected to appear during the first few applications of Stage 3 (Network Decomposition), being grouped into convenient and meaningful **subsystems** at higher levels. As a rule of thumb, the number of **components** at any one level of decomposition should be not more than twelve.

Stage 3 Network Decomposition

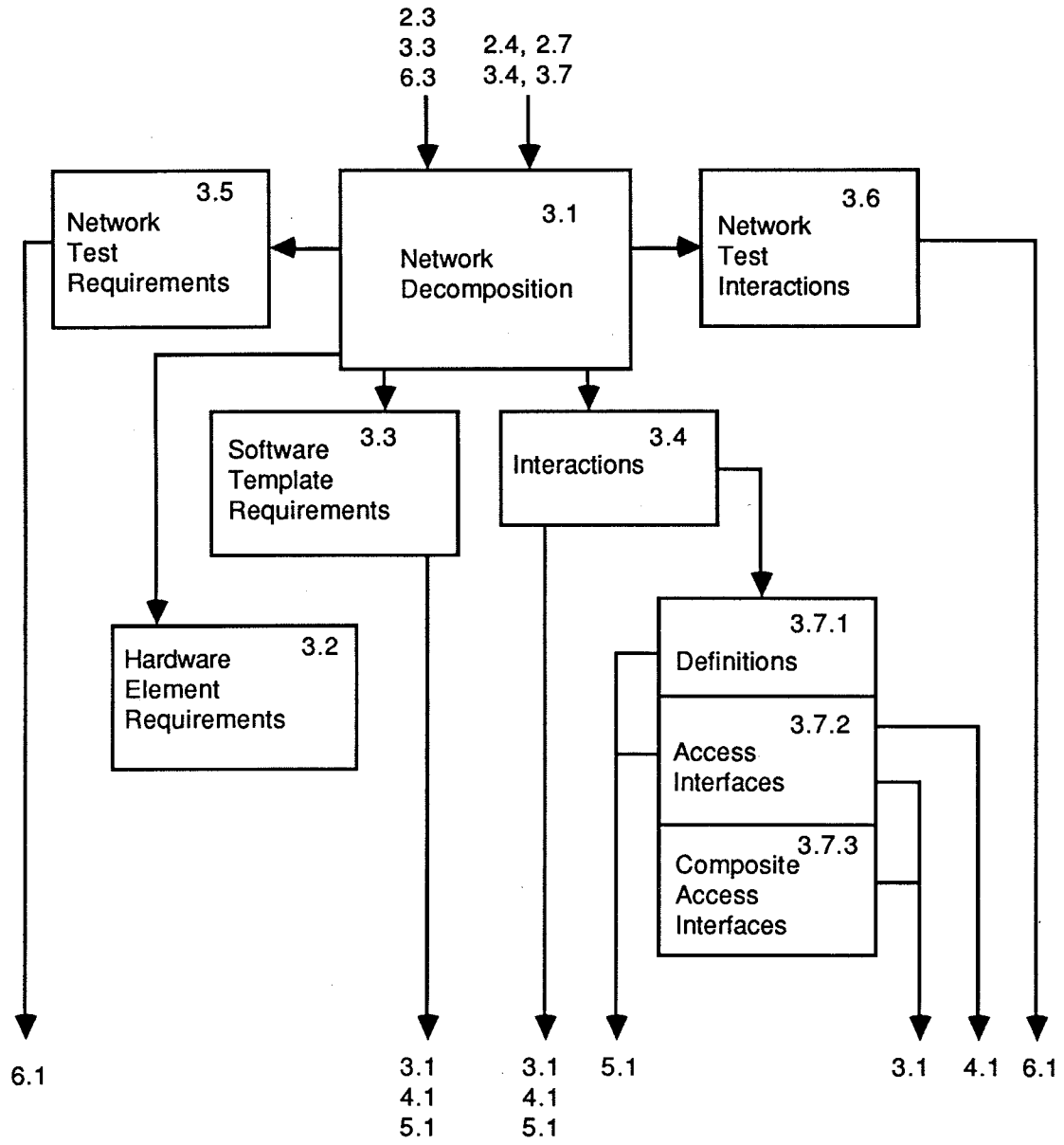
This stage is concerned with the progressive decomposition of a **network** in terms of lower level **networks** and **elements** and is illustrated diagramatically below. It is applied initially to each **network template** which has resulted from the application of stage 2 (inputs from 2.3, 2.4 and 2.7) and to any **network templates** identified as necessary for testing purposes (input from 6.3 in stage

6 described later). Subsequently, it is re-applied to every lower level **network** which results from the decomposition of a higher level one (inputs from 3.3, 3.4 and 3.7). This process is continued until the design is expressed purely in terms of **elements**. The whole stage may be omitted if stage 2 has not produced any **components** which are **networks**.

Stage 3.1 (Network Decomposition) describes a network in terms of its components.

Stage 3.2 (Hardware Element Requirements) describes the characteristics of each hardware element identified at this stage of the design. Hardware components at this and lower levels can be regarded as embedded in the software design.

Stage 3.3 (Software Template Requirements) describes the **template** requirements for each component of the **network** design which is to be implemented by a Mascot **component**.



Stage 3

Stage 3.4 (Interactions) describes the nature and purpose of the internal interactions (**paths**) between **network components**.

Stage 3.5 (Network Test Requirements) describes the way in which the **network** is to be tested and identifies a test **system**..

Stage 3.6 (Network Test Interactions) describes the particular test data, expected results and the operator control sequences required to test the **network**.

Stage 3.7 (Internal Interaction Elaboration) identifies the **definitions** (3.7.1), **access interfaces** (3.7.2) and **composite access interfaces** (3.7.3) required to describe the internal **paths** and the types of the data which flow along them.

Stage 3 carries on the **network** decomposition process started in stage 2. It differs from stage 2 only in that it is totally concerned with internal design decomposition and does not address the problem of matching the software design to the external environment. This results in two significant differences. First, the driving input for stage 3 emerges from previous Mascot design work (stage 2, stage 3 and stage 6 (where the decomposition process is being used for test **network** design)). Second, a formally defined test **system** is required to test a stage 3 **network**; hence the output to stage 6 (note that stage 3.5 will identify the name of the test **system** for subsequent elaboration in stage 6).

As for stage 2, the result of the initial attempt, at this stage, to identify a **component** as a **subsystem** (to 3.1), or an **element** (to 5.1) is necessarily provisional. It is also possible that subsequent decomposition fails because design constraints cannot be met, and in that case the current stage must be repeated.

Stage 4 Element Decomposition

Stage 4 is concerned with the decomposition of an **activity** in terms of **subelements** and, if necessary, of **subelements** in terms of lower level **subelements**. **IDAs** and **servers** are precluded from single thread decomposition in terms of **subelements**. This form of **activity** decomposition is an alternative to conventional program structuring techniques (for example, local procedures, blocks etc.). It has the advantage of producing a software structure visible in terms of Mascot diagrams and of increasing the potential for re-usable **modules** by the linking of separately compiled units. It allows selective servicing of the **paths** which are connected to the enclosing **element** or **subelement** and generally improves testability.

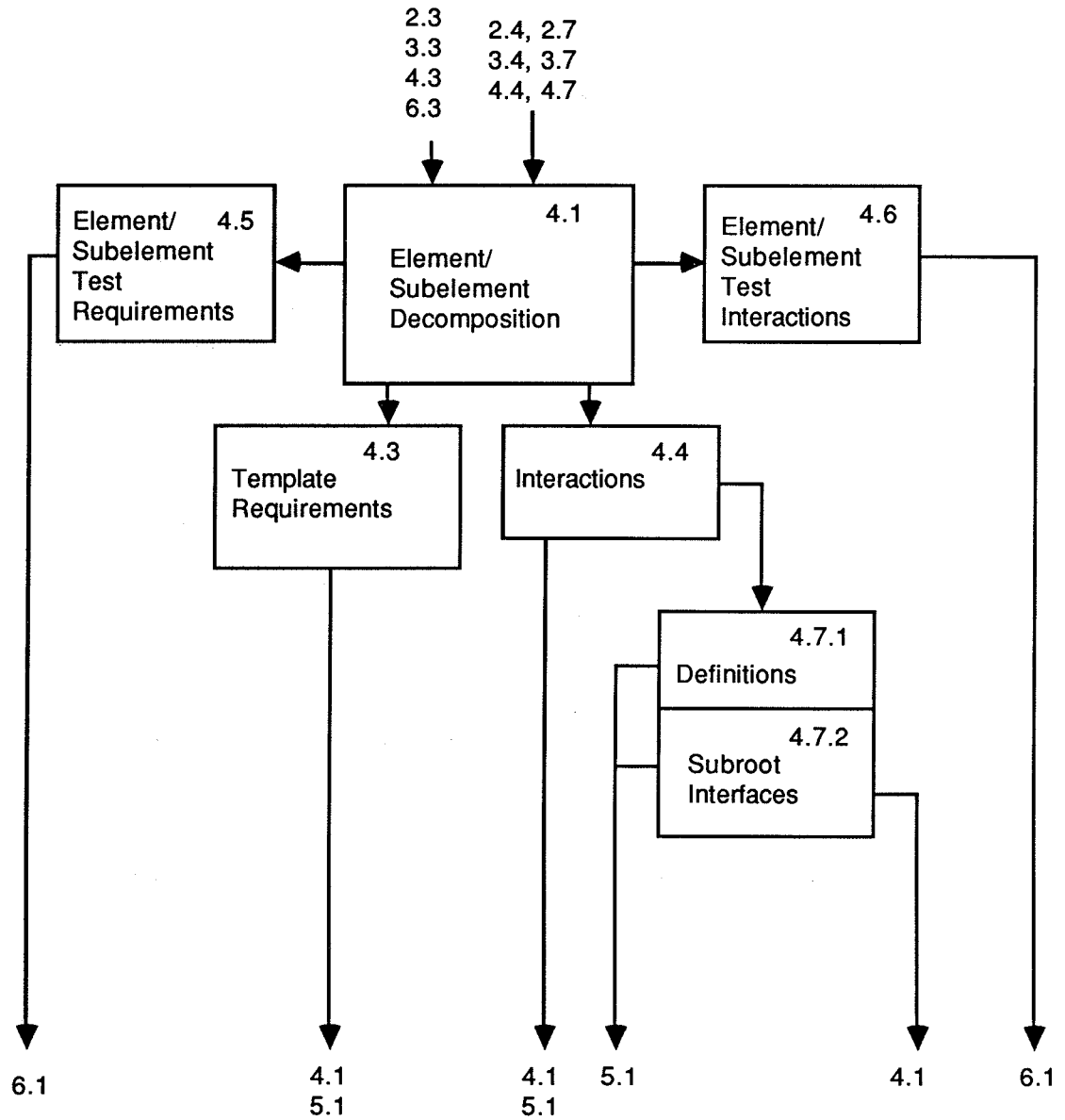
Stage 4 is illustrated diagramatically below. Stage 4 is applied initially to each **element**, chosen for decomposition, which has resulted from the application of stage 2 or stage 3 (inputs from 2.3, 2.4 , 2.7, 3.3, 3.4 and 3.7) and to any similar **element** identified as necessary for testing purposes (input from 6.3

in stage 6 described later). Subsequently, it may be re-applied to lower level **subelements** which result from the decomposition of a higher level **element** or **subelement** (inputs from 4.3, 4.4 and 4.7). This process is continued until the design is expressed purely in terms of **simple subelements**. The whole stage may be omitted if stages 2, 3 and 6 do not produce any **activities** which need to be decomposed.

Stage 4.1 (Element/Subelement Decomposition) describes an **element** or **subelement** in terms of **subelements**.

Stage 4.3 (Template Requirements) describes the **template** requirements for each **component**.

Stage 4.4 (Interactions) describes the nature and purpose of the internal interactions, (that is **links**) between **components**.



Stage 4

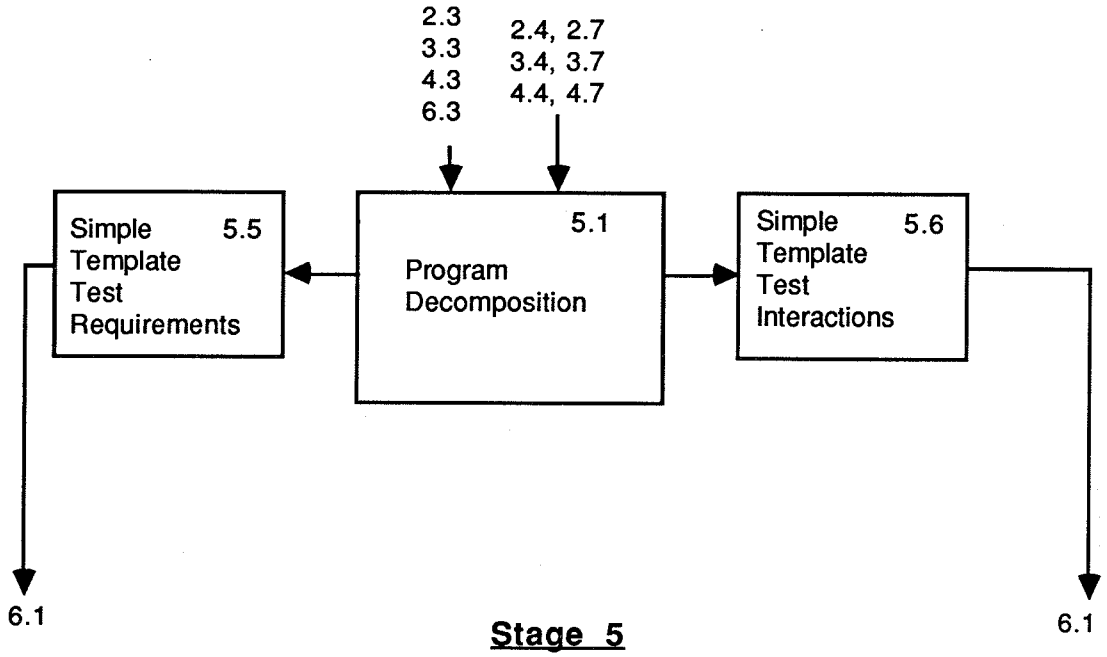
Stage 4.5 (Element/Subelement Test Requirements) describes the way in which the **element** or **subelement** is to be tested.

Stage 4.6 (Element/Subelement Test Interactions) describes the particular test data, expected results and the operator control sequences required to test the **element** or **subelement**.

Stage 4.7 (Interaction Elaboration) identifies the **definitions** (4.7.1) and **subroot interfaces** (4.7.2) required to describe the internal **links** and the types of the data which flow along them.

Stage 5 Program Definition

Stage 5 is the programming stage during which the source text for each **simple module** is produced. Its diagrammatic representation is:



Stage 5

Simple templates, produced in any of the previous 3 stages, form the input to stage 5 (from 2.3, 2.4, 2.7, 3.3, 3.4, 3.7, 4.3, 4.4 and 4.7). Further **simple templates**, needed for testing purposes, may be derived from stage 6 (6.3) which is described later.

Stage 5.1 (Program Decomposition) describes the design of **simple templates** in terms of algorithms and data structures.

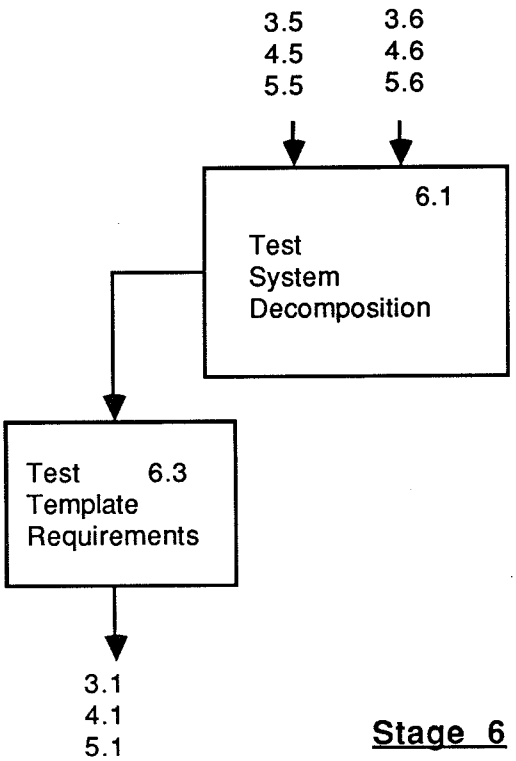
Stage 5.5 (Simple Template Test Requirements) describes the way in which the **simple template** is to be tested.

Stage 5.6 (Simple Template Test Interactions) describes the particular test data, expected results and the operator control sequences required to test th**simple template**.

If during the program decompostion it is found that further structural decomposition would be desirable then stage 5 may be abandonned and stage 3 or stage 4 invoked, as appropriate. Thus stage 5 terminates the primary design process as far as Mascot is concerned, although it may result in a requirement for supporting work on test **system** design. Any conventional technique for program design may be used within this stage provided it is used consistently. However, its use should be confined to the expression of algorithms and data structures within a single **module**; decomposition in terms of separately compiled modules is regarded as a stage 4 design action and should use the appropriate Mascot design feature. If the functional requirements cannot be met within the constraints then the previous stage must be re-invoked.

Stage 6 Test System Definition

Stage 6 is concerned with the integration and testing of application software. Its diagrammatic representation is:



Stage 6

Input to stage 6 consists of the test requirements and interactions for **networks**, **elements/subelements** and **simple templates**, respectively, identified in stage 3, stage 4 and stage 5 (from 3.5, 3.6, 4.5, 4.6, 5.5, and 5.6).

Stage 6.1 (Test System Decomposition) describes the structure of the test system.

Stage 6.3 (Test Template Requirement) describes what each test **template**, required to create a**component** specifically identified for test purposes, has to do.

Stage 6 is concerned with the first level of test **system** decomposition. The requirements for **networks**, **element/subelements** and **simple templates** identified in stage 6.3 are fed back to stages 3, 4 and 5, respectively for further decomposition as necessary. Notice that if a standard harness can be used for testing, and if no special test **components** are required, then stage 6.3 may be omitted. Stage 6 designs will work entirely in terms of the substage 6 test interactions and substage 4/substage 7 operational interactions derived from the previous stages.

Status Progression

Throughout all six stages of the development the Mascot design notation is used to record the product of the design process, and the **status** progression facilities are used to record progress and to control the use to which the various parts of the design definition may be put. Mascot formality is limited to these aspects but any other formal techniques for function or data flow definition may be used in conjunction with Mascot with direct reference to **templates**, **components**, **interfaces**, **paths** or **links** in the design structure.

5.2 DOCUMENTATION

Introduction

The use of the Mascot method should be supported by a suitably structured approach to the documentation requirement at each stage in development. Well-structured documentation will support the system throughout its life-cycle by relating system design to requirements and external constraints, allowing descriptions to be created and maintained in a recognisable framework and facilitating access to this information by the wide variety of personnel who need it.

It is not intended in this Handbook to provide prescriptive techniques or standards for documentation, because there is a wide range of diverse documentation standards to which Mascot systems will be required to conform. However, it should be recognised that software documentation does not exist in a vacuum and must be part of the overall documentation strategy. The documentation structure for a medium-to-large project is inevitably large and complex and involves many inter-dependencies. It is essential that this structure is planned at the outset of a project and used by the development team as the project progresses. Used correctly, a recognised structure will reduce duplication/overlap in documentation, facilitate access to information, allow consistency checking and localise the effects of software amendments on documentation.

The following points should be taken into account when planning the documentation structure:

- The Mascot method assists in re-usability of software, and the re-usability of associated documentation is an important consideration. As far as possible, information which is project-specific should be isolated, in documentation terms, from re-usable information.

- A single **template** can be used to create several **components** in a typical Mascot application, and so information concerning the creation of **components** should be separated from the formal software descriptions and documentation of the **template**.

- The development and target environments for embedded software, for example tools and hardware configurations, are liable to change in the potentially long life (say, 20 years) of a typical Mascot application. The documentation should lend itself to such amendment by separating information describing the environment from formal software description.

The primary aim of documentation is to aid comprehension. Without compromising this aim, it is possible

to use the Mascot **module classes** as a framework for re-usable documentation. Wherever feasable, description should be restricted to the scope of the object being documented. For example, the documentation of a **template module** should avoid reference to the functionality of elements beyond the **interfaces** which it provides and requires.

The result of this approach is documentation which is both abstract and localised in nature. Although this achieves the goal of re-usability, there is a danger that over-zealous abstraction may lead to documentation which is disjoint, making it difficult to comprehend overall functionality without reference to numerous items of documentation. It can be seen that the goals of comprehensibility and re-usability are not always compatible.

As a general rule, the characteristics of a description should be similar to those of the object being described. If a **module** is created for widespread general use then re-usability of documentation is of paramount importance. If a **module** has many dependencies on the characteristics of external objects (say, for efficiency reasons), then the requirements of comprehensibility must take precedence. Most software is a compromise between the desire for re-usability and the needs of a specific application, and so engineering judgement is required when determining what form of documentation should be used for individual **modules** and **module classes**.

Template Documentation

By way of illustration of the above, consider the requirements for the documentation of an individual **template**. The documentation must describe the functionality of the **template** and the **interfaces** which it provides and requires (the specification documentation). If the **template** is **composite**, the documentation must identify the internal structure of the **template** in terms of **components** and their inter-connectivity (the implementation documentation).

In all cases, the descriptive content of the specification documentation must be detailed, complete and unambiguous. This is the lowest level in the hierarchy at which this documentation will be presented.

The more difficult aspect of **template** documentation concerns the level of detail of the implementation documentation. Clearly, the rationale behind the decomposition within the **template** must be presented in full. The level of description given for the **components** and their inter-connectivity must be carefully considered.

It would be possible to identify the **components** and internal **interfaces** by name only, and refer the reader to lower level **modules** whose specification documentation will provide the information on the functionality of individual items. Whilst this would meet the goal of re-usability, such an approach would tend to be abstract to the point of obscurity. Alternatively, if detailed descriptions of the means by which internal **components** achieve their functions are provided, then this leads to undesirable duplication

within the documentation hierarchy.

Experience indicates that the preferred approach in most circumstances is to describe in full how the **template** achieves its required function, and to provide brief descriptions of the **components** and their inter-connectivity, referring the reader to lower levels of the documentation hierarchy if more detail on a particular **component** is required. This approach allows a reader to assimilate sufficient concise information to understand how the internal elements of a **template** achieve the necessary functionality of that **template**, without unnecessarily duplicating the specification documentation of the lower level **modules**.

The Mascot method encourages isolation of information by recognising the independent nature of active and passive elements, **interfaces**, **definitions**, **libraries**, etc. As far as possible, without losing sight of the need for comprehensibility, the documentation strategy should follow the same approach.

Control of Documentation

In a project of any reasonable size, a large quantity of documentation must be administered and controlled. The documentation is the primary method of communicating information about the structure and status of the system under development to all interested parties in the project. As the system is developed and amended, so, too, is its associated documentation. It is clearly necessary for the documentation to be in step with the system it describes at all stages. Generally, this can be achieved for documentation in the form of in-line comment, because this is retained within the **module** it describes and is subject to the configuration management and quality control procedures applied to individual **modules**.

Some project documentation is not directly related to a specific **class** of Mascot **module** (for example, requirements, quality reports, design rationale documents). Nonetheless, it is strongly recommended that all project documentation is held in a machine-readable form under the same configuration management database as that used to administer software development. Wherever possible, documentation relating to a specific **module** should be held as part of that **module**. Although this does not guarantee that documentation will be updated in line with the software, its proximity will encourage good practice and ease consistency checking.

The configuration management database can be used to create and maintain a knowledge of the dependencies between documents and **modules**. When a **module** or document is updated, the dependency relationships can be used to identify other **modules** or documents which may be affected by the change. **Status** progression is the minimum facility provided by the Mascot **database**, and will generally deal with identifiable Mascot **module classes**. In some cases, it may be possible to provide similar mechanisms for documents.

If, for some reason, documentation is not held in a machine-readable form and related in a controlled fashion to the system being described, procedures must be adapted to provide confidence that documentation and software will proceed through development in a consistent manner. Configuration management and quality assurance procedures must be applied to the documentation, even if only by manual means.

Access to Documentation

The exisistence of documentation does not in itself guarantee its accessibility. The sheer extent of documentation can make it difficult to identify and retrieve precisely that part of the documentation which is necessary to understand a particular facet of the system and carry out an item of work. The problem of collating and constraining relevant information is not new, however, and it is important to provide such a facility if work is to be carried out efficiently.

The publication of, and adherence to, the overall project documentation structure will assist in the identification of relevant information. However, in many cases the information needed to build a clear picture of the context in which a piece of work is to be carried out will be distributed through many **modules**. These **modules** will also contain material which is extraneous to the individual's requirement. There is a strong case, when employing Mascot, to consider the use of a documentation tool which will assimilate the specification documentation and implementation documentation of a number of **modules** into a single document. This will allow a reader more easily to comprehend the role of a **template** in the overall system. The existence of such a tool may reduce the temptation to duplicate documentation at different levels of the document hierarchy.

The tool would be achievable by identifying (say, by a keyword mechanism) the specification documentation and the implementation documentation within each **module**. In order to see more clearly the role of, say, a **subsystem**, a user potentially could gather together into a single document:

(a) the implementation documentation of the enclosing **subsystem**,

(b) the specification and implementation documentation of the **subsystem** under scrutiny and

(c) the specification documentation of the **components** and **interfaces** which comprise the **subsystem**

using the relationships between the **modules** and the documents established by means of the configuration management database.

Purpose of Documentation

There are two major categories of documentation according to its prime purpose: documentation of the project and documentation of the resultant system. These are now discussed in turn.

Documentation of the Project

Documentation is created and amended throughout the system life-cycle in order to capture and express the current status as development proceeds. However, it is not sufficient merely to maintain current status information. The development of documentation through the system life-cycle reflects the design decisions and rationale which invariably affect and shape the path to the final system. It is important, therefore, to capture and maintain the documentation as it existed at various major points in the system's evolution. If an 'audit trail' of this form can be created, it will allow analysis of the design process which led to the complete system. Further, if this information can be allied to quality records, modification records, etc., then analysis can prove extremely valuable to future projects and can be exceptionally useful during maintenance and support of the system.

In many cases, collation of the information required for this analysis is difficult as it exists in different forms (some as part of **modules** in the host system, paper records held by the quality assurance department, etc.), and results in an incomplete and disjoint record. The use of a suitable configuration management tool (possibly allied to the Mascot **database**) incorporating design change control and quality assurance recording mechanisms will assist in collating information in a manner which is coherent and provides a convenient form of reference.

Documentation of the System

The descriptive information required to support the resultant system incorporates not only the documentation of the **templates** and associated **modules** used to create the **network components**, but also the documentation of the testing mechanisms used to verify functionality. The relationship between documentation of these areas requires careful organisation. There is not necessarily a one-to-one correspondence between test **systems** and **templates**. However, there must be a mechanism for relating the **template** documents to the test **system** documentation, preferably one which does not adversely affect the re-useability of the documents.

The documentation of a **template** may refer to a test **system** created to test that **template**. It should not refer to test **systems** which test sub-**networks** in which the **template** is used. Test **system** documentation should refer to all **components/templates** in the sub-**network** under test. The result is that the test documentation structure reflects the **network** structure and the documentation of the sub-**networks** should refer to the test **systems** used to verify functionality.

General Documentation Requirements for Module Classes

There are generalised requirements for the documentation of all **modules** in a Mascot **system**, as outlined below.

The following categories should be considered in the documentation of all **modules**, although some categories may not be applicable directly to a particular **module class**:

Language dependencies
Environment dependencies
Machine dependencies
Code characteristics (eg conditional compilation, assembler inserts)
Test status
Quality assurance status
Test mechanism

For each **module** under development, the following must be available, either embedded in documentation or controlled by tools:

Version number
Modification audit trail
Mascot **database status**

All **modules** must refer to, or include in-line, the requirement which they are designed to meet.

5.3 SYSTEM TESTING

Introduction

During the design and development stages of a project there are several techniques which have traditionally been used to establish confidence in the software being created and to demonstrate that it meets its requirements. These include formal techniques for proving correctness, design reviews, in which, among other activities (see Section 5.1), program source text is subjected to the collective scrutiny of the project team, and testing, in which the behaviour of the corresponding executable code is systematically exercised through the use of test data. Testing is, and is likely to remain, the most widely used of these techniques. This is partly because of its wider scope including , as it does, the effects of production tools, operating system, hardware and external environment, and partly on account of the greater psychological assurance which it engenders ('seeing is believing'). It is also, at the present time, the only practical method.

Mascot, in common with other software development methods, contains a graphical form of design representation which can be exploited to advantage in carrying out a design review. It is in the sphere of testing, however, that Mascot provides significantly greater assistance than most other methods. In particular the modularity scheme of Mascot, together with the unique **template-component** model, constitutes a powerful support mechanism for testing. In addition, the ability to supply a separate **system template** for each test **network** provides a sound basis for the configuration management of testing. The **template/component** model makes it possible to create test **systems** which can co-exist with the application **systems**. This greatly simplifies the handling of regression testing during maintenance.

Provided that suitable run-time environments are available, and that the application **templates** contain no hardware specific features, the test **networks** may be executed on either host or target systems (see Section 3.3).

In this section the general considerations for the testing of Mascot **systems** are discussed and a specific testing strategy is proposed.

General Considerations

The testing of any substantial piece of software, from a large sequential program to a full blown system involving concurrency on a large scale, needs to be carried out in a modular fashion. There are two widely recognised approaches to this known respectively as top-down and bottom-up.

In the top-down approach the highest level unit is produced first and 'stubs' are created to replace the units at the next lower level. These stubs are sections of program which normally perform some simple

actions such as signalling the fact that they have been evoked and reporting the values of any input data with which they have been supplied. In some cases, however, a stub may partially emulate the actions of the unit in whose place it stands. Testing involves examining the pattern of invocation of the lower level entities. Stubs are then progressively replaced by the corresponding application units accompanied, where necessary, by a further level of stubs. The great merit of this approach is that testing and integration proceeds in a series of roughly equal steps. The last step is no more significant in itself than any of the others and no more likely to reveal any fundamental defect. Also application units can be constructed using their stubs as a starting point.

In the bottom-up approach it is the lowest level units which are produced first. Specially written drivers are then used to test them, first individually, and then in larger and larger groups until the entire application has been built. New drivers are needed at each stage and discarded after use. The final step in this process is highly significant as it may show that the overall structure is incorrect. Despite these disadvantages the bottom-up approach is widely used in practice and is considered by many programmers to be the natural way to test software.

While a strong case can be made out, at least in theory, for application of the top-down approach to testing sequential programs, it is more difficult to see how to apply it to the networks of concurrent processes required to solve real-time problems. In Mascot, therefore, its application is likely to be limited to the testing of the implementation details of **simple activities**, **roots**, **subroots** and **libraries**. The testing of Mascot **networks** is likely to be carried out bottom-up.

In devising a general strategy for testing Mascot **modules** it is necessary to cater for three groups: **specifications**, **simple templates** and **composite templates**. Since **specifications** have no direct realisation in the executable software there is no direct testing associated with them.

Simple templates constitute the algorithmic building blocks of the system, and the testing of these **templates** is primarily aimed at detecting algorithmic and programming errors. For this purpose a mixture of 'black box' test data, based on the design specification, and 'white box' test data based on the actual design is appropriate. At the very least these latter data should exercise every sub-condition of each conditional expression so as to ensure that every statement in the program is executed at least once.

Composite templates are groupings of **components** derived from **simple** and **composite templates**. The primary aim in testing them is to detect errors of communication between the **components** and to verify the real-time behaviour.

An Example Testing Strategy

This section describes a specific testing strategy developed from one that has been used successfully in several projects which employed an earlier version of Mascot. It supports testing of both **simple** and

composite templates and serves here as an example of recommended practice. It was designed to meet the following major requirements :

1. The test strategy and supporting tools should support both formal (that is Quality Assurance) and informal (that is development) testing in a consistent and reproducible manner.

2. The production and maintenance of the test data should require a minimum of special tools.

3. The tools should support interactive diagnosis with data entered from the terminal and results displayed at the terminal.

4. To facilitate regression testing, the performance of tests and the interpretation of results should require a minimum of manual intervention.

5. Since in many cases the target hardware does not become available until late in the software production cycle, it should be possible to perform the majority of the testing on the host computer.

The strategy is based on the concept of a test script containing, in a machine-readable form, the directives necessary to initialise a test system, data which are to be presented to the unit under test and results which are to be expected if the unit is correct. A test script is prepared in a textual form which may be manipulated by means of standard text editors. If used directly in this form, execution time is taken up, during the test, in performing the necessary type conversions. An alternative is to employ special tools to convert the script, separately, to a binary form but this was not adopted in view of general requirement 3 above.

A test script may be interpreted by a standard **network** of test-administering **components**, supported by application specific type conversion **components**. These two sets of **components**, together with the unit under test, form a test **system**.

The precise nature of the conversion **components** is dictated by the communication requirements of the unit under test. For testing **IDAs**, **servers** and **libraries** an **activity** is the natural choice since such **components** need to be driven. A **root** can most readily be used to convert data for a **subroot** which is under test, while for testing **roots** and **activities** an **IDA** may be chosen as the conversion **component**.

The role of the standard test **components** is to provide textual information, from a test script, to the conversion **components**. The latter convert this information to a suitable form and transmit it to the unit under test. This process is reversed in handling the responses. These are taken from the unit under test by the conversion **components**, transformed into text and passed to the standard test **components** which generate the results.

Thus, each conversion **component** refers, in general, to three **interfaces** describing, respectively, text input, text output and communication with the unit under test. The textual **interfaces** are usually **ports** which may be connected to **windows** of the standard test **components**. One or other of the text **interfaces** may be omitted in some cases. The **interface** to the unit under test may be an **access interface**, a **subroot interface** or a **library interface**. The Mascot monitoring facilities may be used, during testing, to provide evidence of completness.

An example of a test **network**, in which the conversion **components** are **activities**, is shown on the following page. **Subsystems** have been shaded in this diagram in order to improve readability.

Operation of the Example Test Network

Existing operating facilities are used to assign the input and output (files) to the **servers** ***input_server*** and ***output_server***. When the system is STARTed, each of the conversion **activities** declares its identity to the **IDA** ***text*** and receives in return a unique key. The relevant arrowheads on the **network** connections through which this initialising transaction takes place are omitted from the diagram so as to avoid unnecessary complication. Subsequently, messages directed to a particular conversion **activity** can be obtained by the **activity**, from the **IDA**, by quoting the appropriate identifying key.

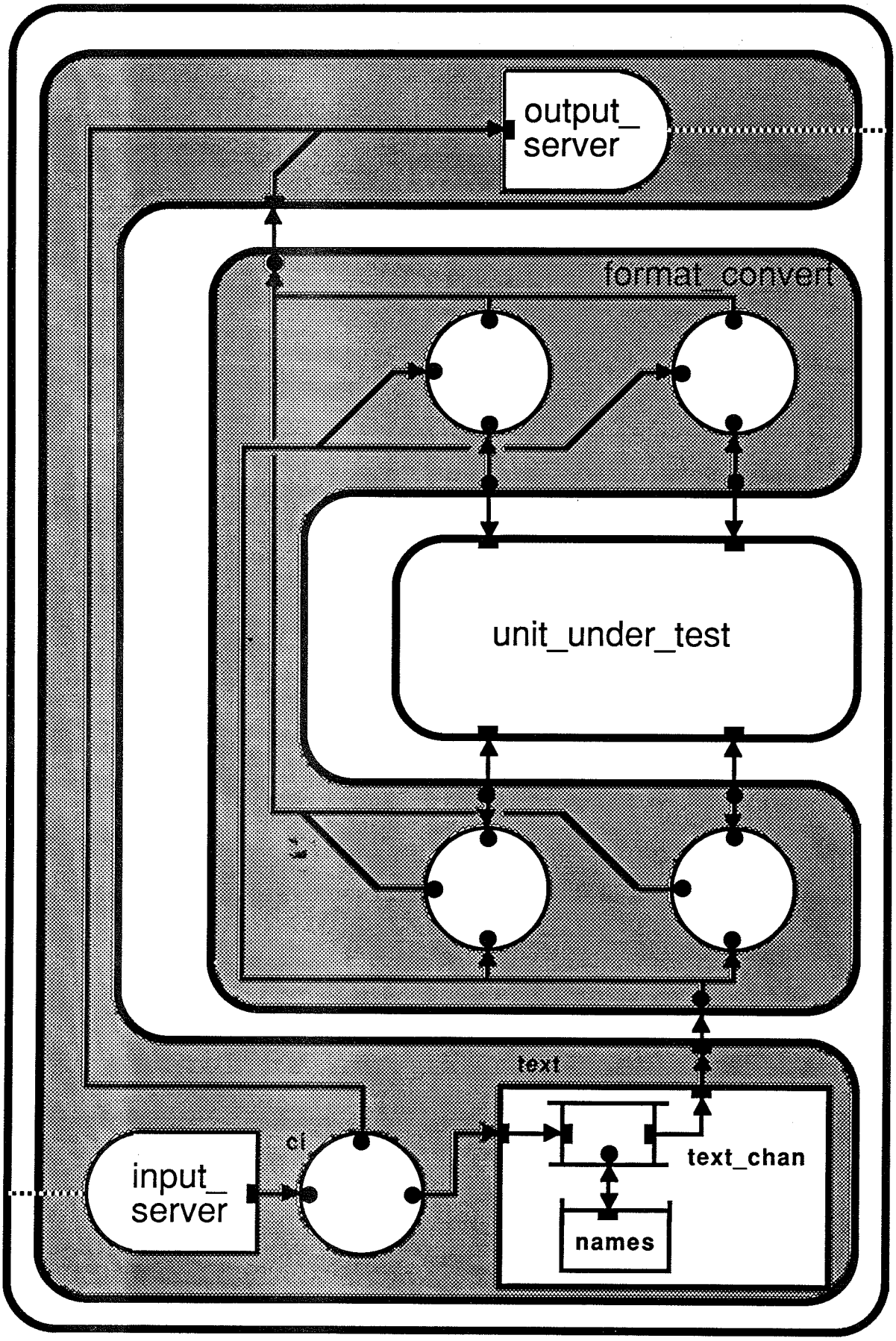
The command interpreter **activity**, ***ci*** , reads a command line from ***input_server*** and copies it directly to ***output_server*** . The purpose of logging the input text together with the results of the test in this way is to allow the sequence of events during the test to be deduced from inspection of the output file. This has been found to work well in practice even when quite large portions of the final **network** are being tested.

The input line is then examined by the command interpreter to see if it contains a command which can immediately be executed and, if so, the required action is performed. Commands whose execution may be initiated by the command interpreter include Execution Control commands, Monitoring Control commmands and special commands to, for example, control the timing of events in the **network**. If the command line cannot be dealt with in this way, it is passed to **IDA** ***text*** where it becomes available to the conversion **activity** whose identity it contains. An error message is generated if the line contains no recognisable identity.

The unit under test in this example may be any passive network fragment, that is one possessing only **windows** on its boundary. Messages received by conversion **activities** from ***text*** are converted to a suitable form and passed to the appropriate access mechanism of the unit under test. Any response is made available at another **window** for use by a converter which processes it for output.

Thus, to handle output, the converters call the appropriate access mechanism, translate the response into textual form and pass the result to ***output_server*** . Under some circumstances it is necessary for output conversion to be controlled from the test script. Hence the connection of all converters in the

diagram to the **IDA *text***. The need for this most commonly occurs when testing a **pool** where the input line may contain a command to read the **pool** contents. However, another possibility is that the input line contains a copy of the expected results which the converter handling output can compare with the actual results and so signal whether the test has been passed or failed.


output_
server
format_convert
unit_under_test
text
input_
server
cl
text_chan
names

APPENDIX A

SYNTAX OF DESIGN REPRESENTATION LANGUAGE

DESIGN REPRESENTATION LANGUAGE SYNTAX

The syntax of the Mascot design representation language is included here for reference purposes. It is presented in two equivalent forms. A set of syntax diagrams, as used for description throughout the body of the Handbook, appears first together with an index. These diagrams follow the conventions established by Wirth in connection with the Pascal programming language. All valid constructs may be generated by tracing all possible paths through each diagram, as indicated by the arrow heads, from the top left until the path terminates on the right. Loops may be repeated as often as required. At each box a basic symbol of the language is generated. Where the box has rounded corners or is circular, the symbol is literally that contained in the box. This has been further emphasised in the former case by the addition of background shading. A rectangular box is a reference to another syntax diagram.

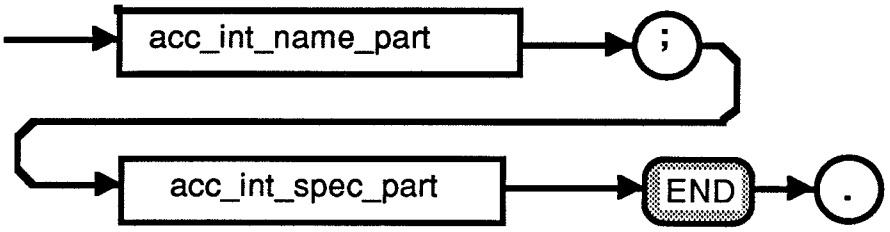
The diagrams are complete, in Mascot terms, in that they include all the non mandatory features. They are incomplete in the sense that they contain a number of undefined symbols whose definition is dependent on the choice of implementation language. Such undefined symbols are indicated by means of a '*' near the top righthand corner of the syntax box. A second group of symbols, for which no defining diagram is included, are marked with a '$'. These are all identifiers; they appear under a variety of names so as to incorporate some semantic information. Where a syntax box is marked with 'An', the corresponding definition diagram appears on page 'n' of the appendix. Where a box is unmarked it represents a symbol which is defined on the same page on which it is used.

The design language syntax is then presented in a metalanguage based on Bachus-Naur notation. In this form it is made specific to the use of Pascal as the implementation language. All terminal construct names are explicitly connected to the Pascal syntax definition. The order and organisation of presentation is such as to facilitate reference to the subsection, in the body of the Handbook, where each construct is discussed. It is followed by an alphabetically ordered list of terms which also contains appropriate subsection references. It should be noted that for B-N notation, as compared with the equivalent syntax diagrams, it has been necessary to introduce additional syntactic categories. Also, category names which in the diagrams have been shortened so as to save space, are given in full in B-N form (for example 'acc_int_spec_part' becomes 'access_interface_specification_part').

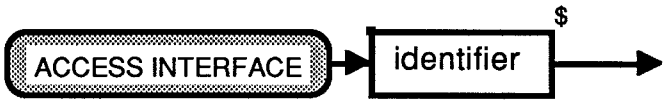
DIAGRAMMATIC FORM OF SYNTAX

Index

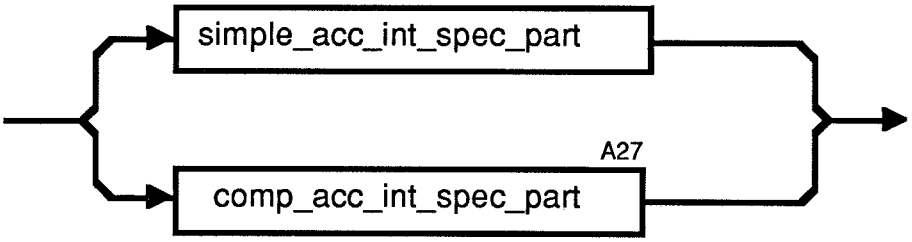
access_interface



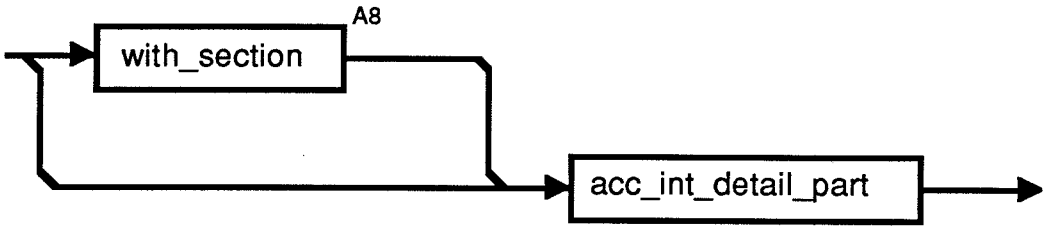
acc_int_name_part



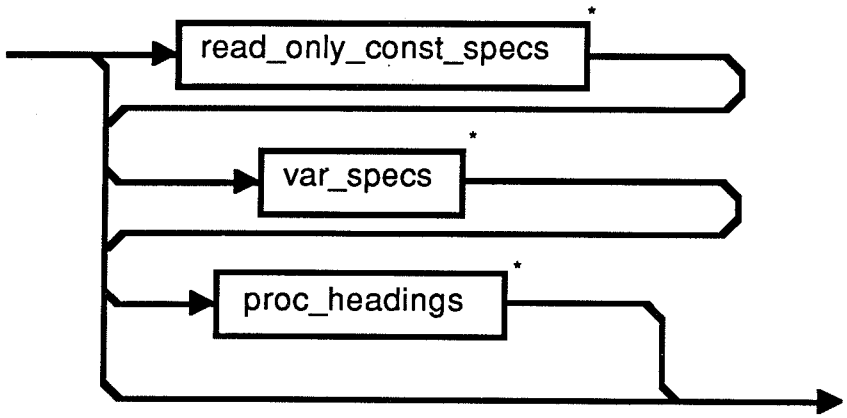
acc_int_spec_part



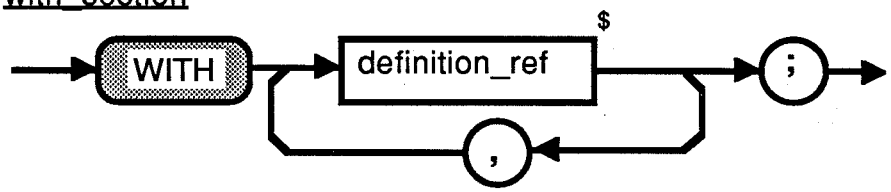
simple_acc_int_spec_part



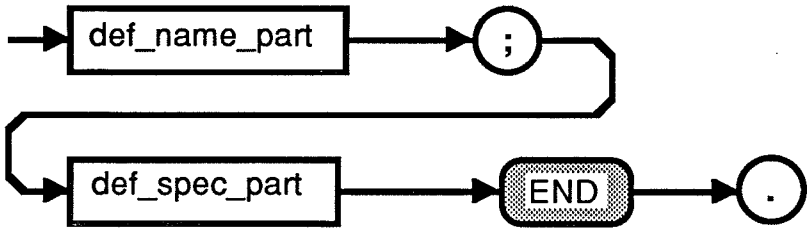
acc_int_detail_part



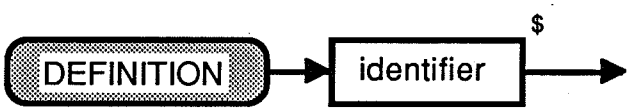
with_section



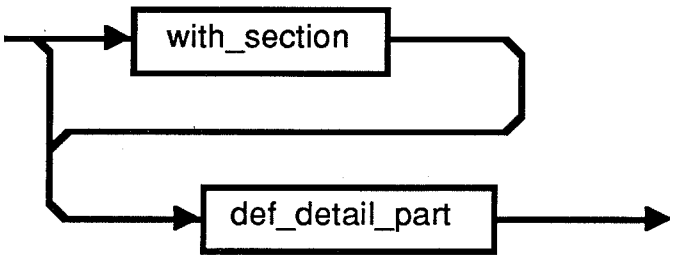
definition



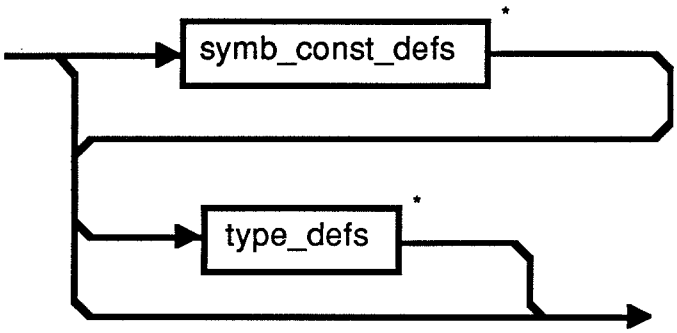
def_name_part



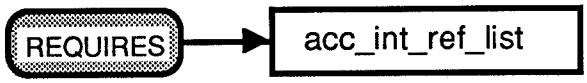
def_spec_part



def_detail_part

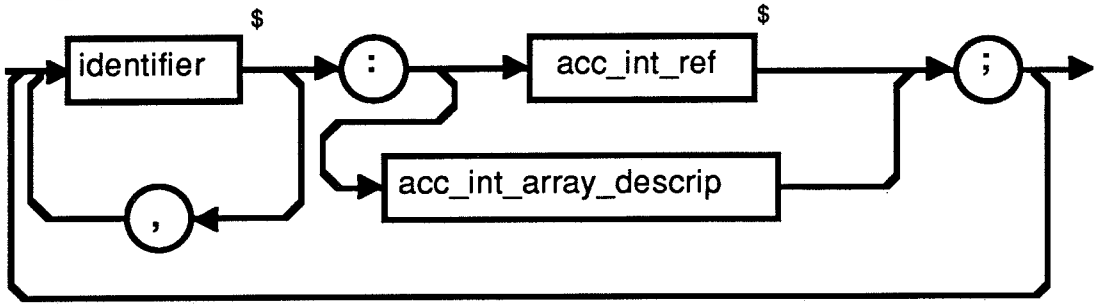


port_spec

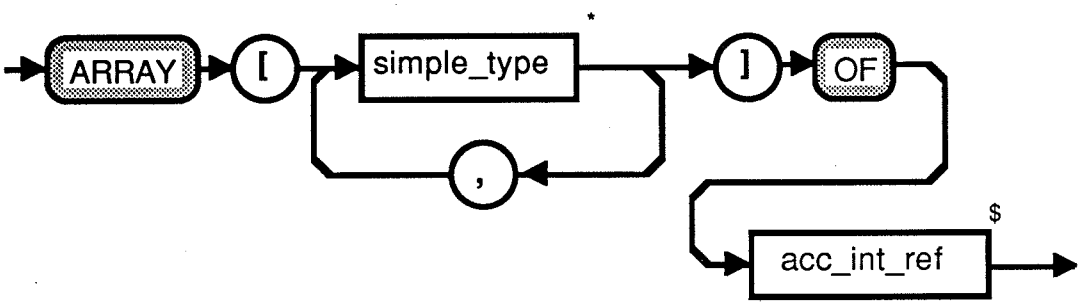


window_spec

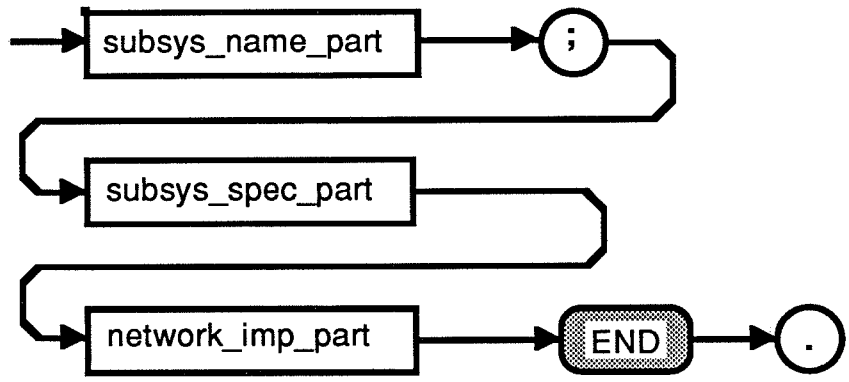
acc_int_ref_list



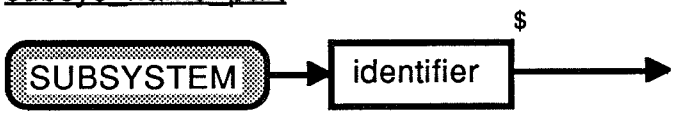
acc_int_array_descrip



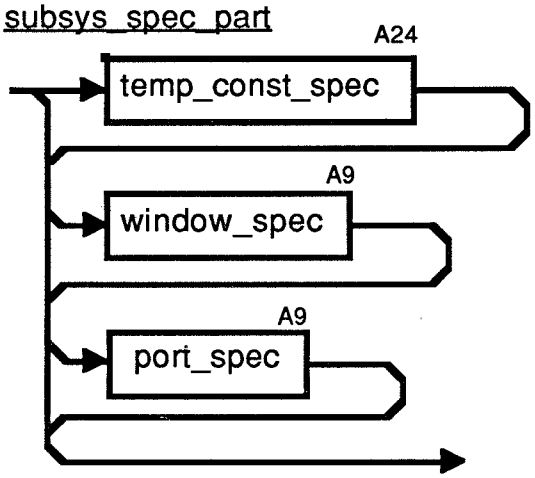
subsystem



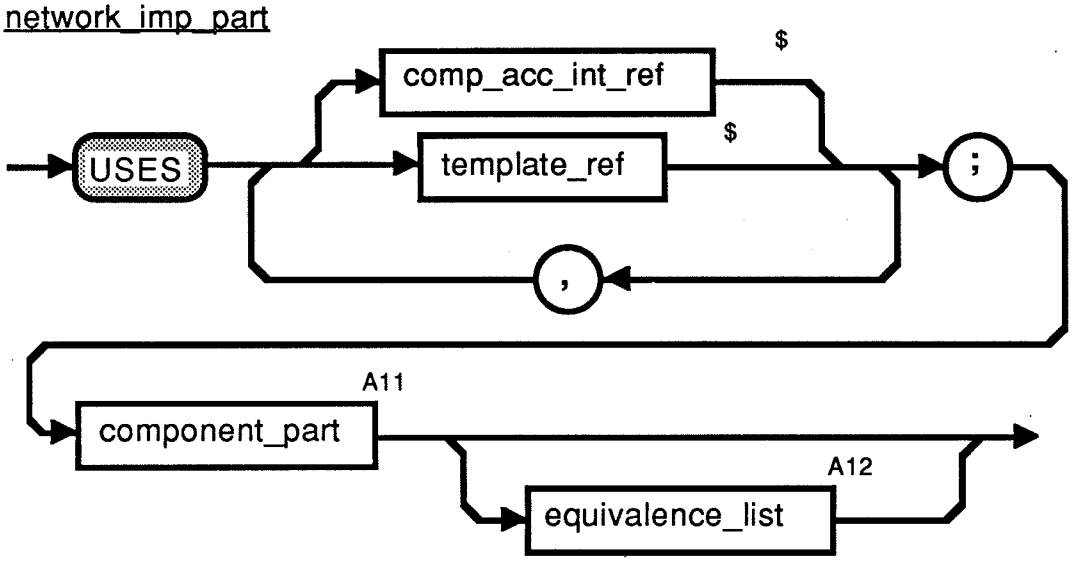
subsys_name_part



subsys_spec_part

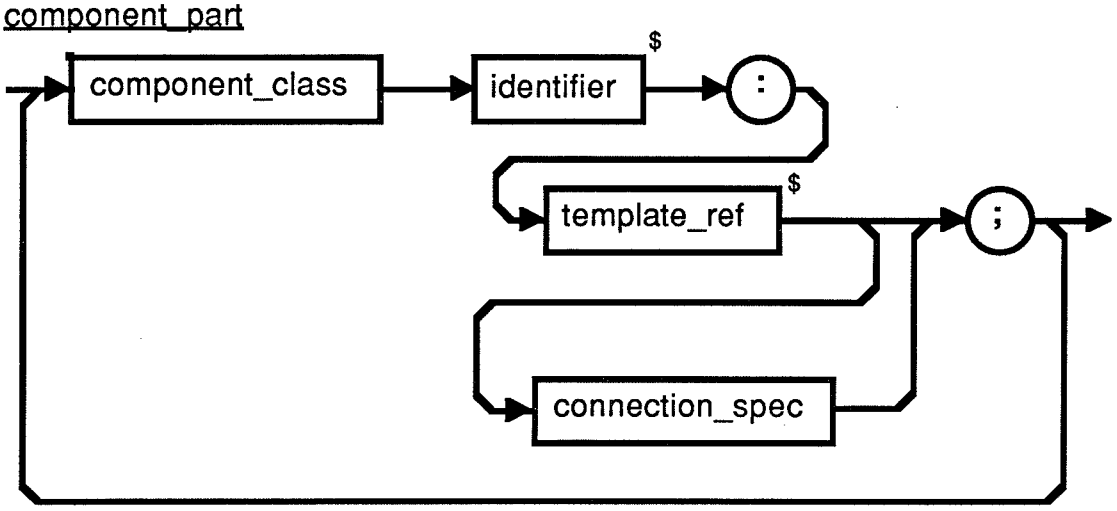


network_imp_part

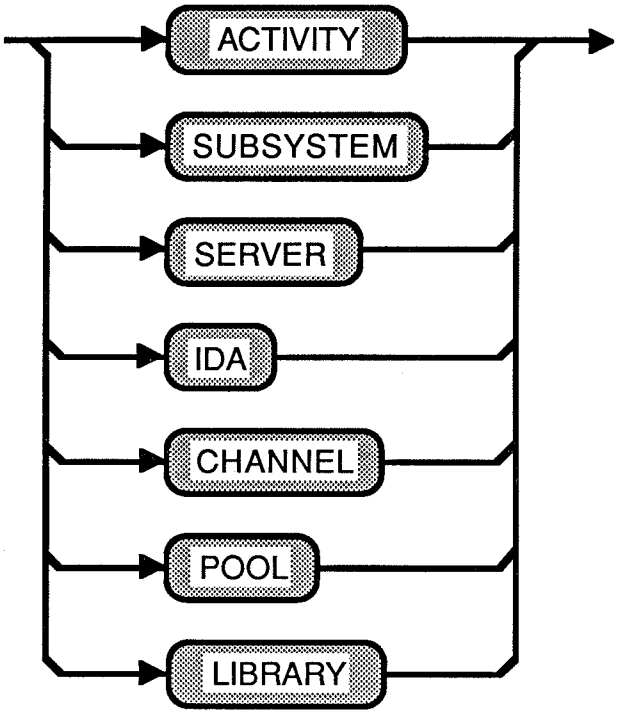


Neither a **composite IDA** nor a **composite server** may refer to a **composite access interface** in its USES list.

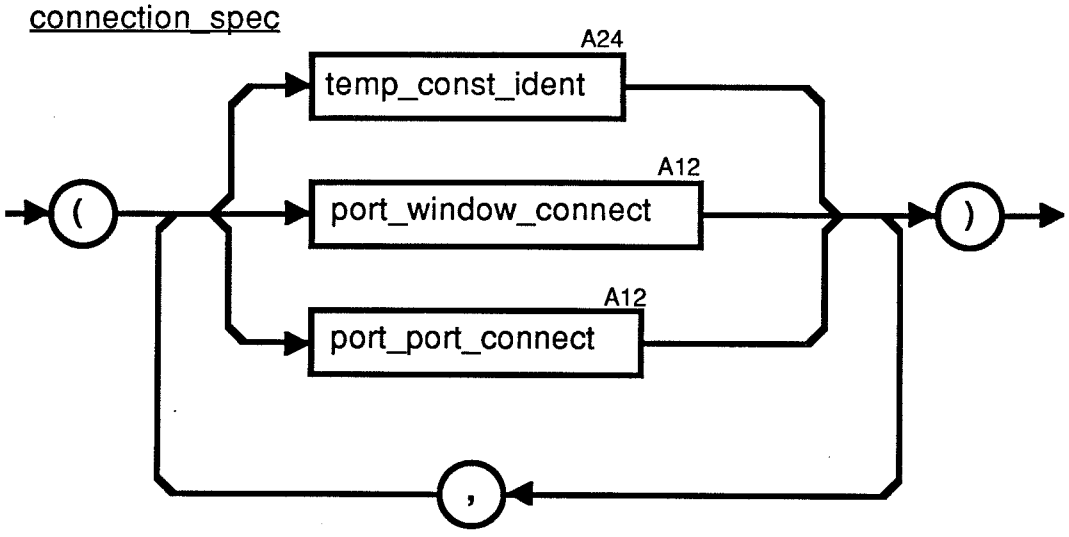
component_part



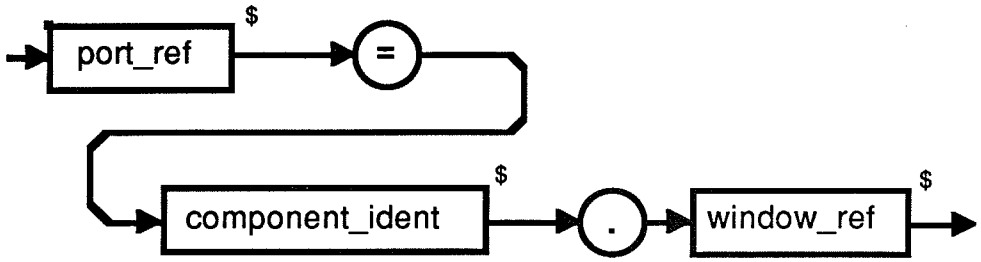
component_class



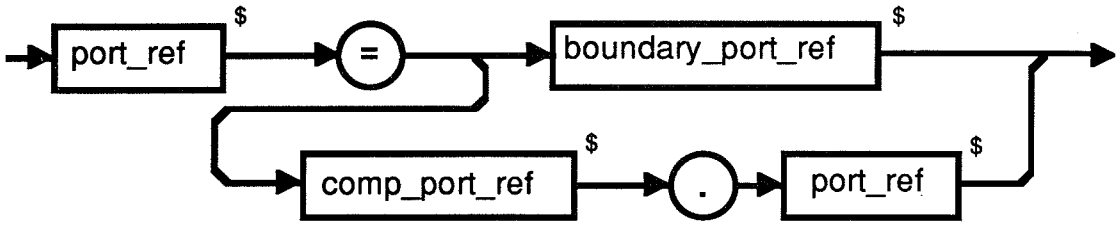
connection_spec



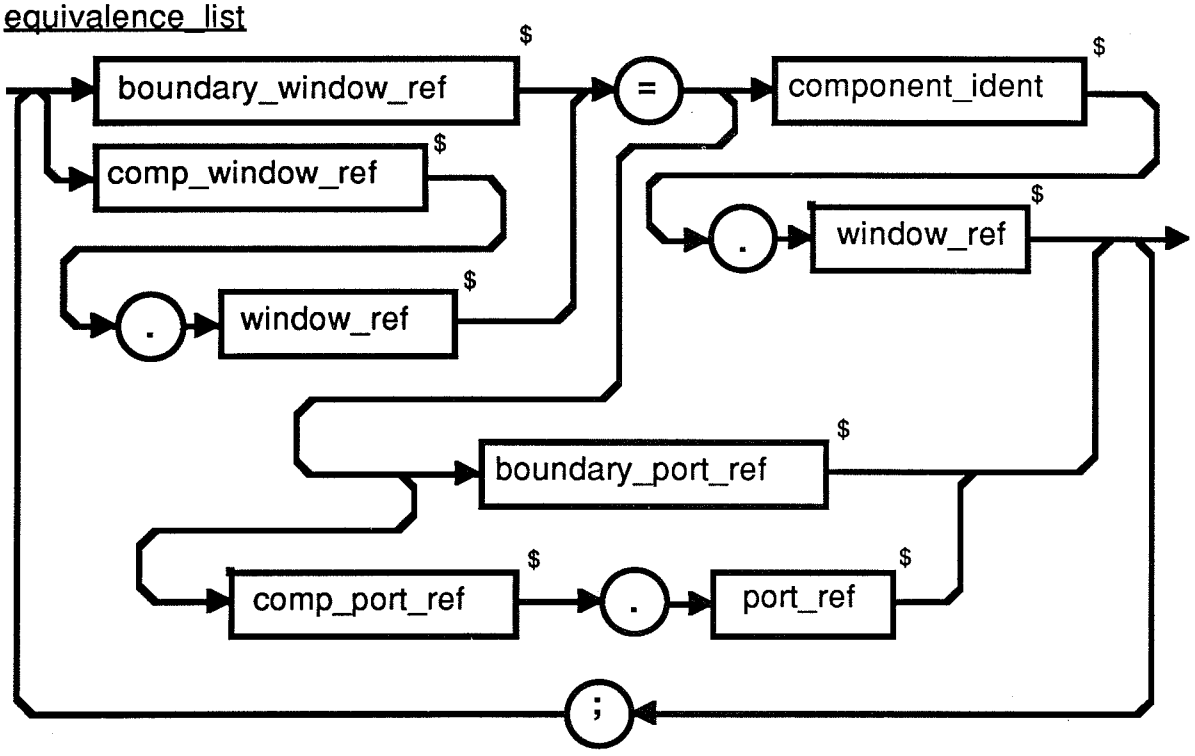
port_window_connect



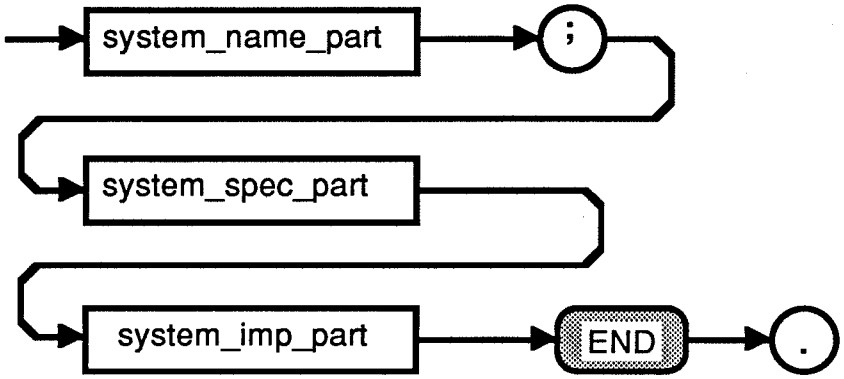
port_port_connect



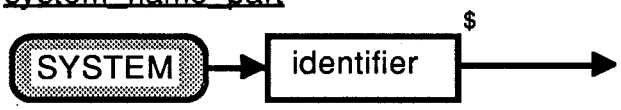
equivalence_list



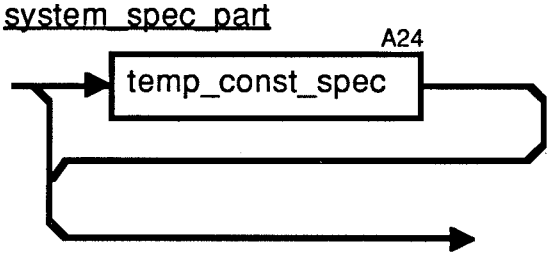
system



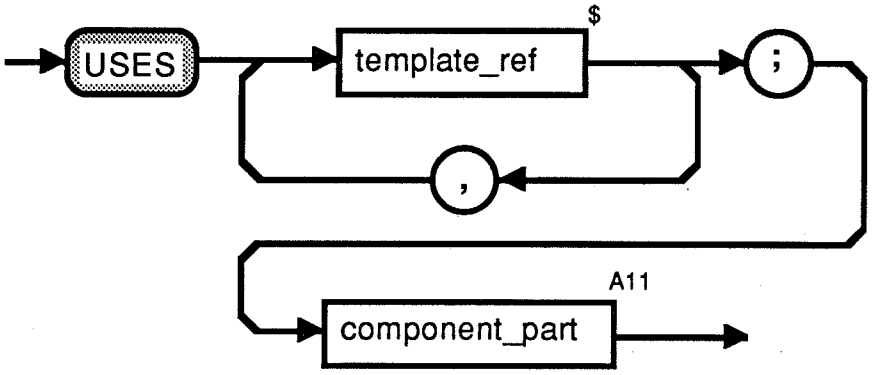
system_name_part



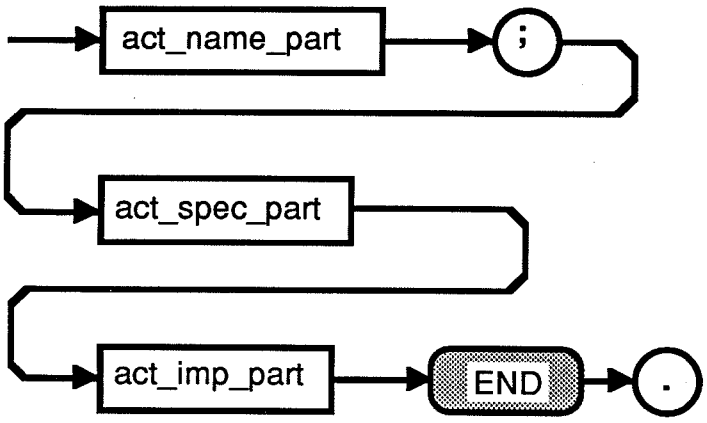
system_spec_part



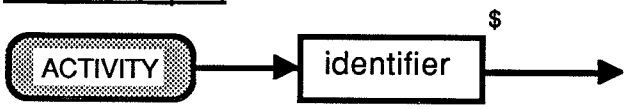
system_imp_part



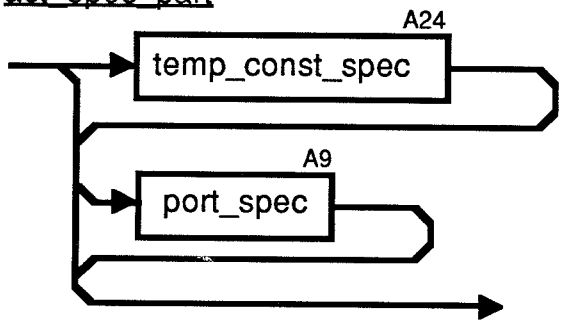
activity



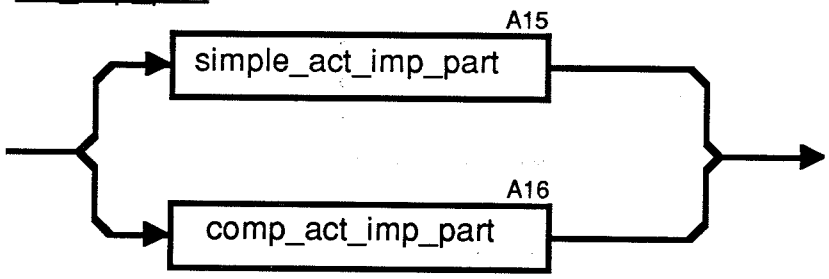
act_name_part



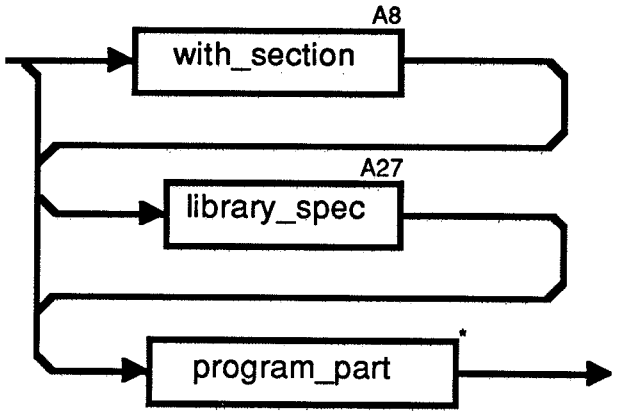
act_spec_part



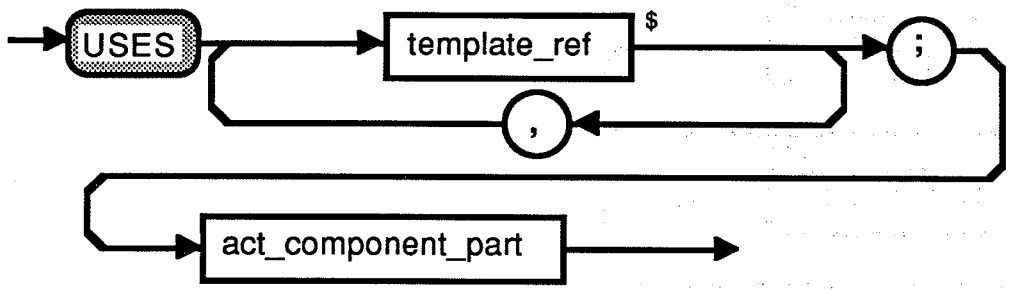
act_imp_part



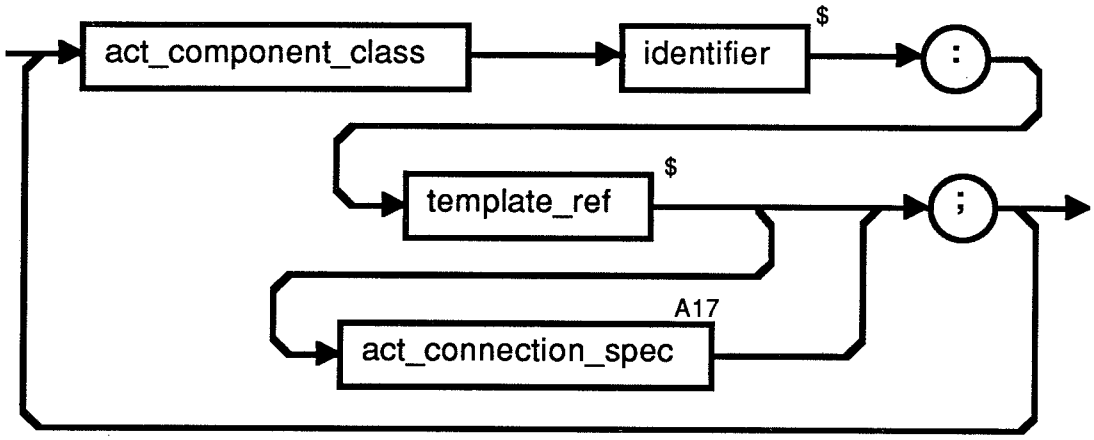
simple_act_imp_part



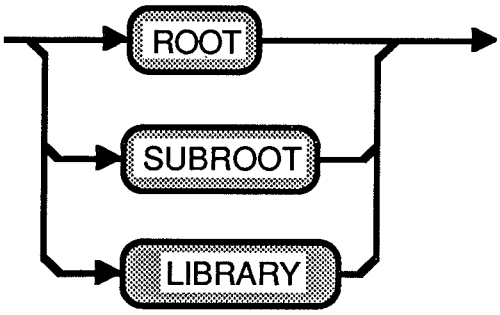
comp_act_imp_part



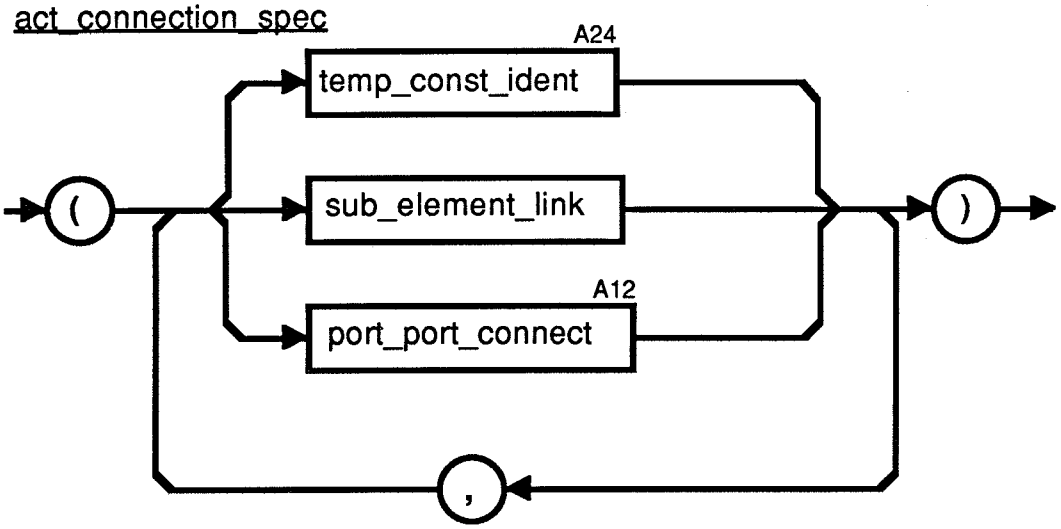
act_component_part



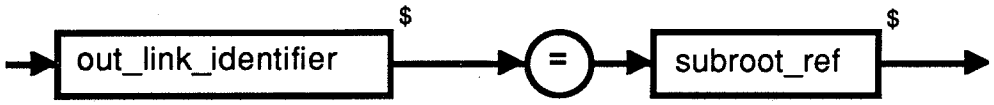
act_component_class



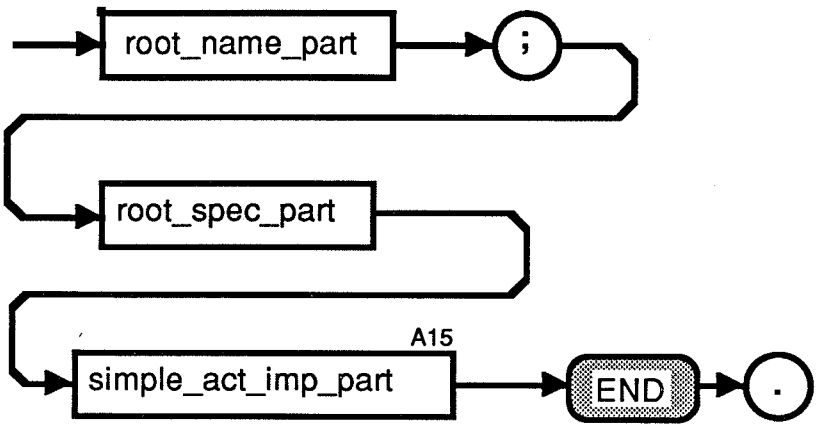
act_connection_spec



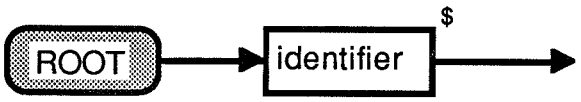
sub_element_link



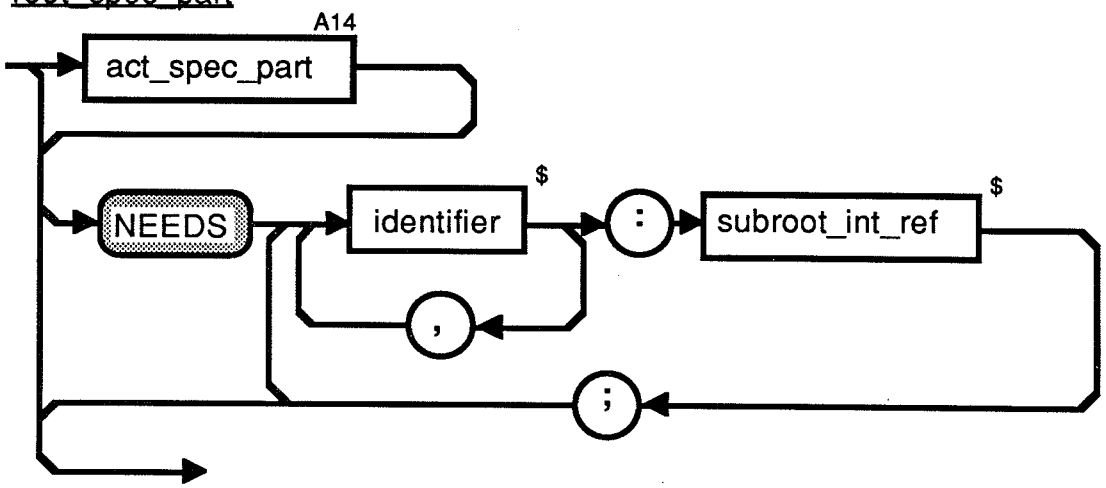
root



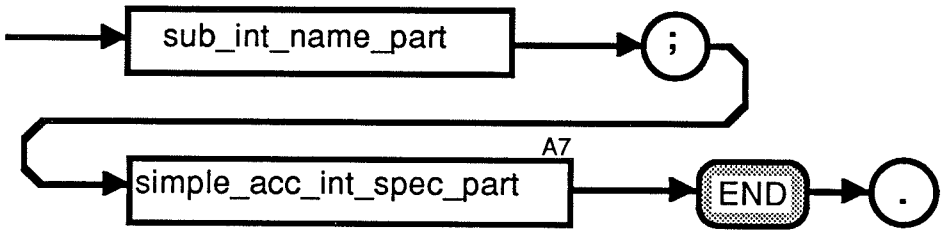
root_name_part



root_spec_part

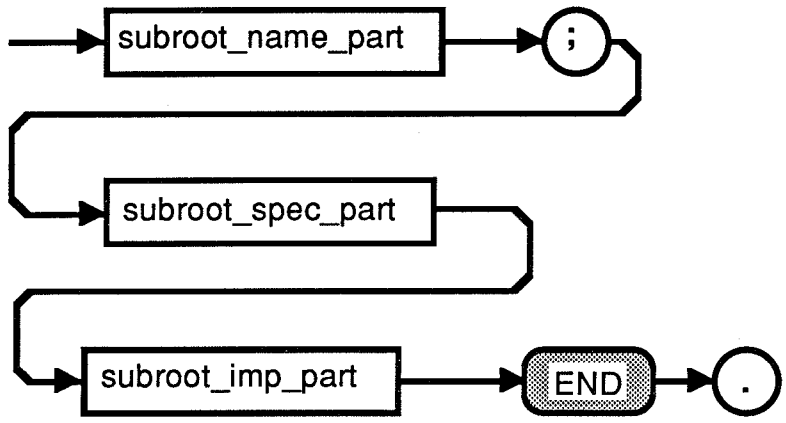


subroot_interface

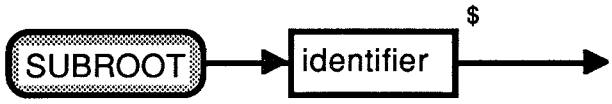


sub_int_name_part

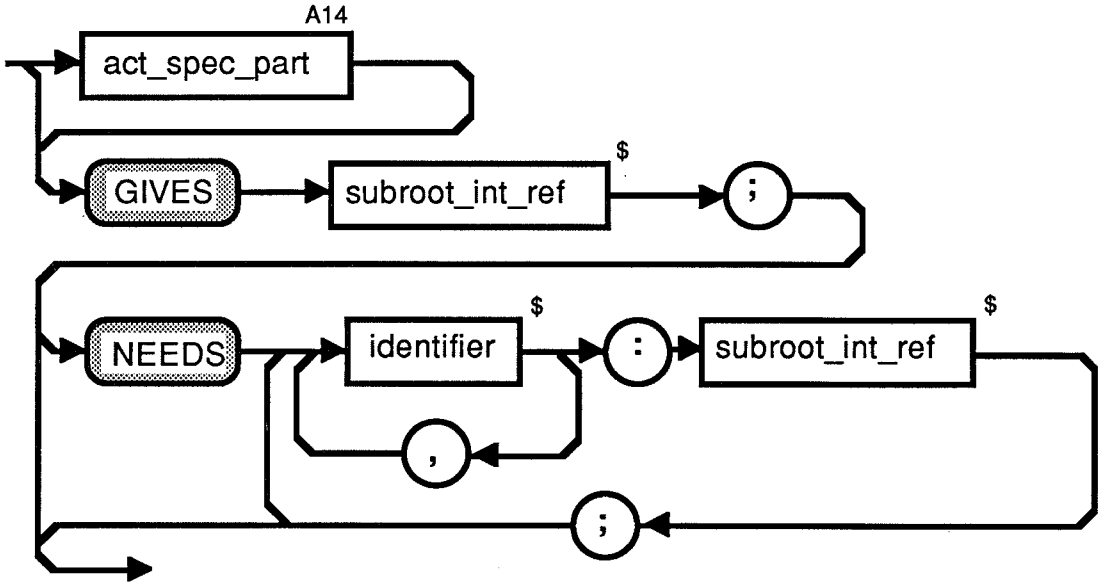
subroot



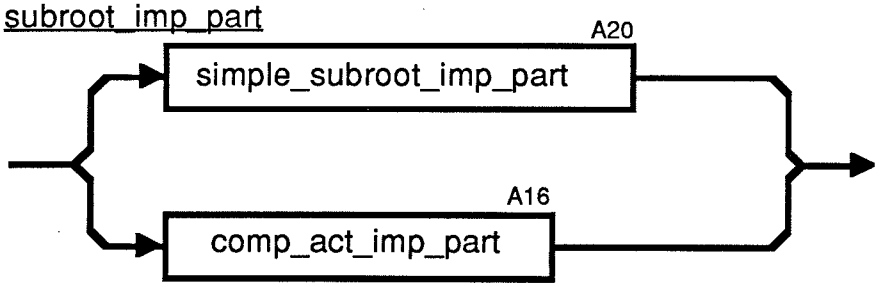
subroot_name_part



subroot_spec_part

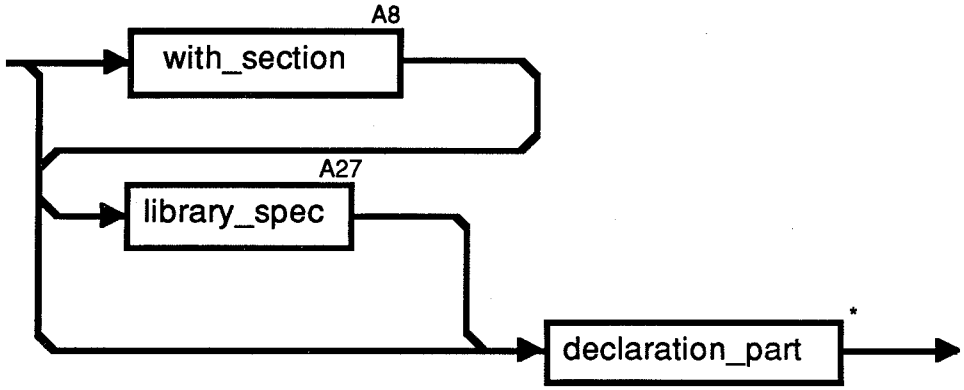


subroot_imp_part

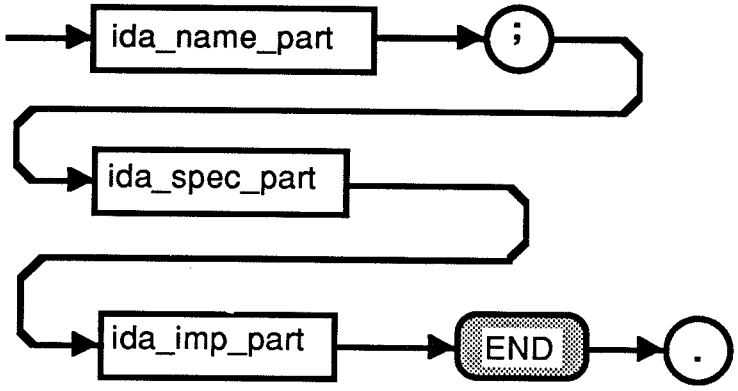


A **composite subroot** may not contain a **component** derived from a **root template** but must contain a **root component** derived from a **subroot template**. This latter **component** gives the **interface** to the **composite template**.

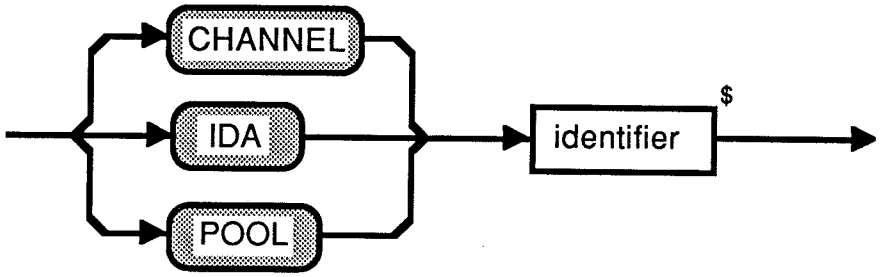
simple_subroot_imp_part



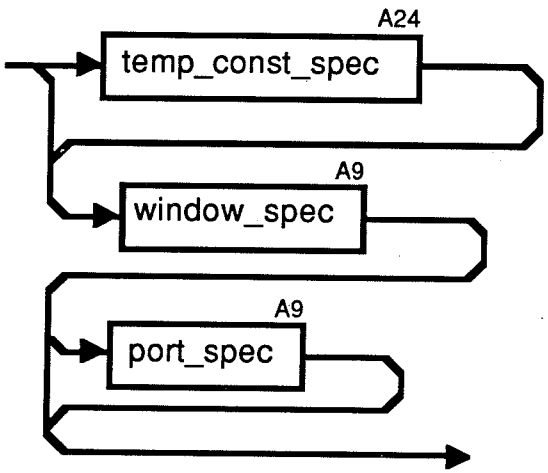
ida



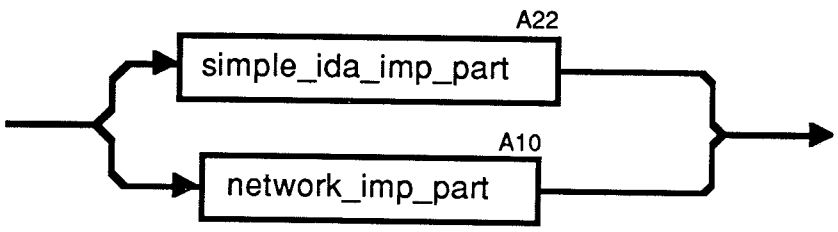
ida_name_part



ida_spec_part

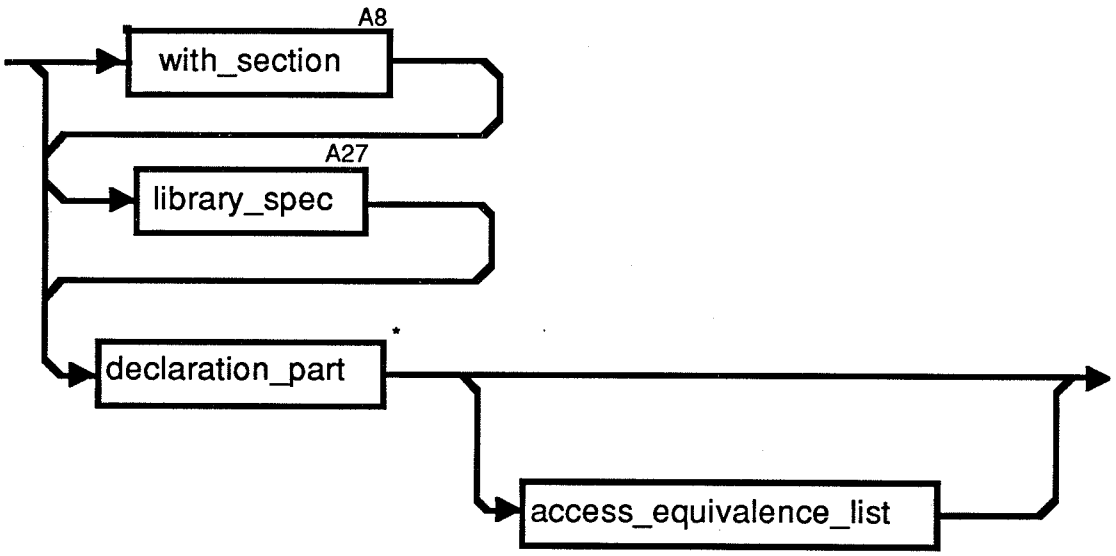


ida_imp_part

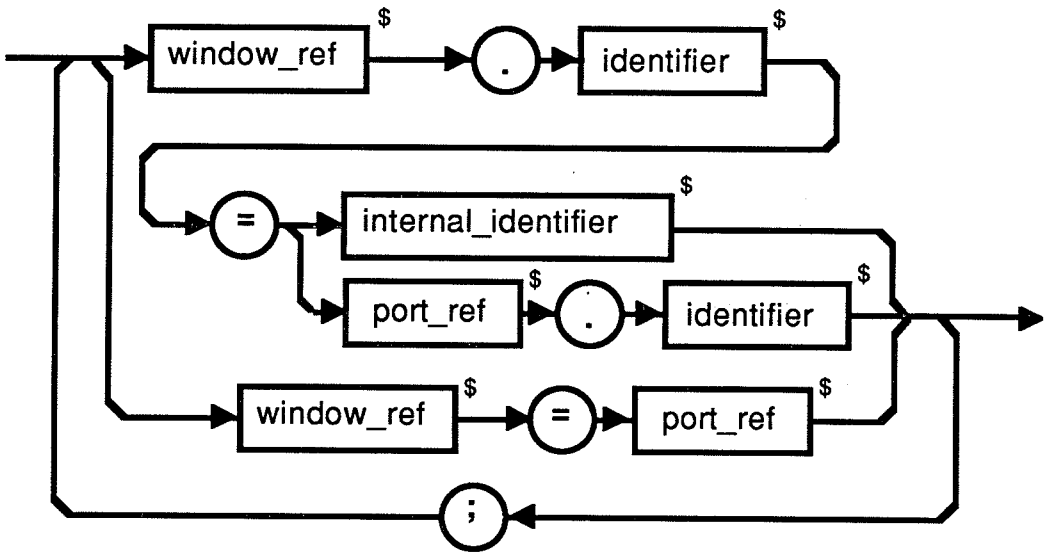


A composite IDA may contain only **IDA, channel, pool** and **library components.**

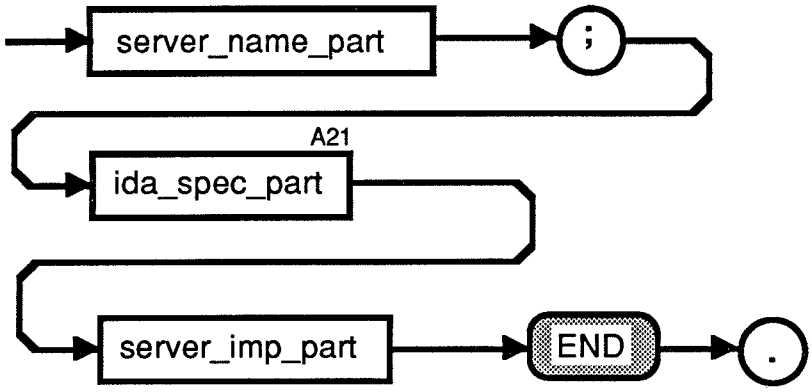
simple_ida_imp_part



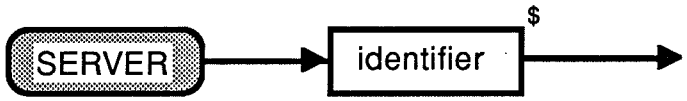
access_equivalence_list



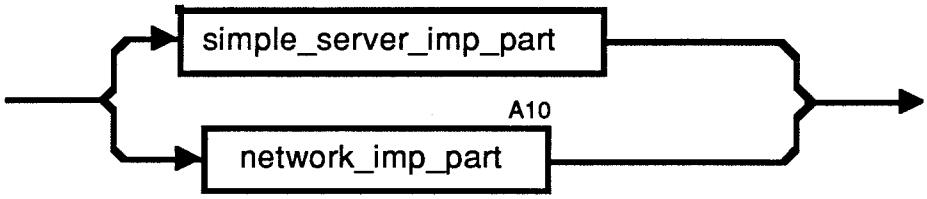
server



server_name_part

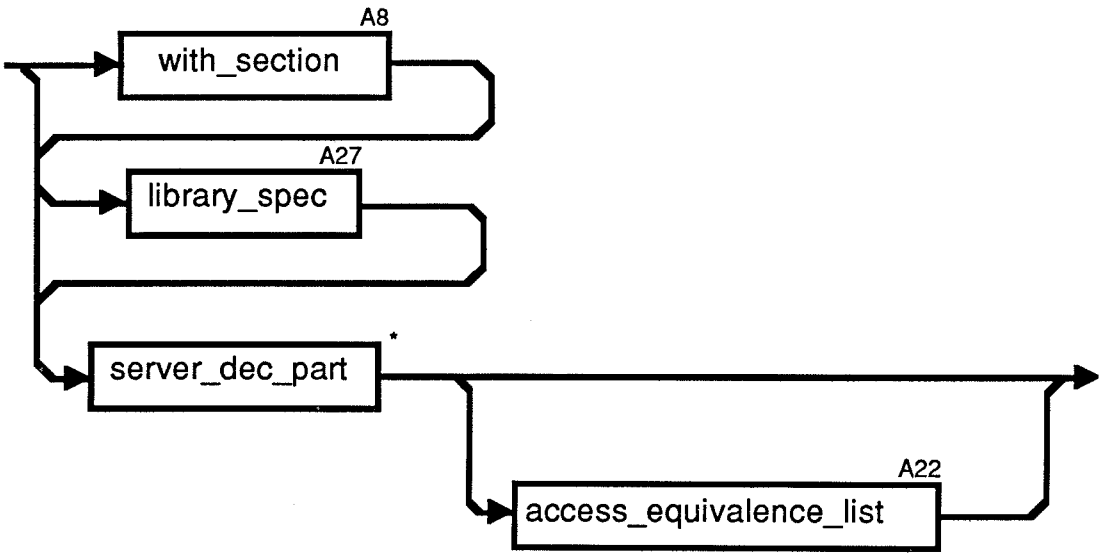


server_imp_part

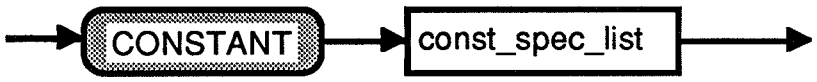


A **composite server** must contain at least one **server component** and may also contain **IDA, channel, pool** and **library components.**

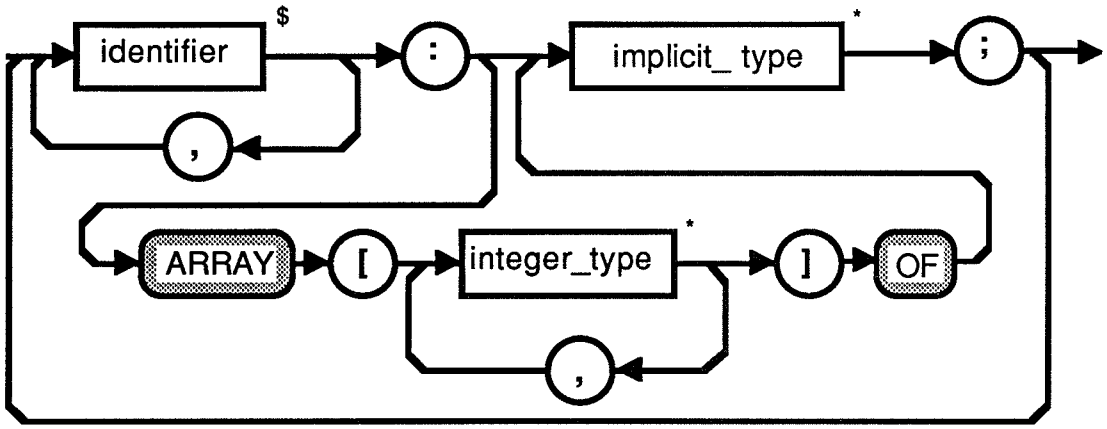
simple_server_imp_part



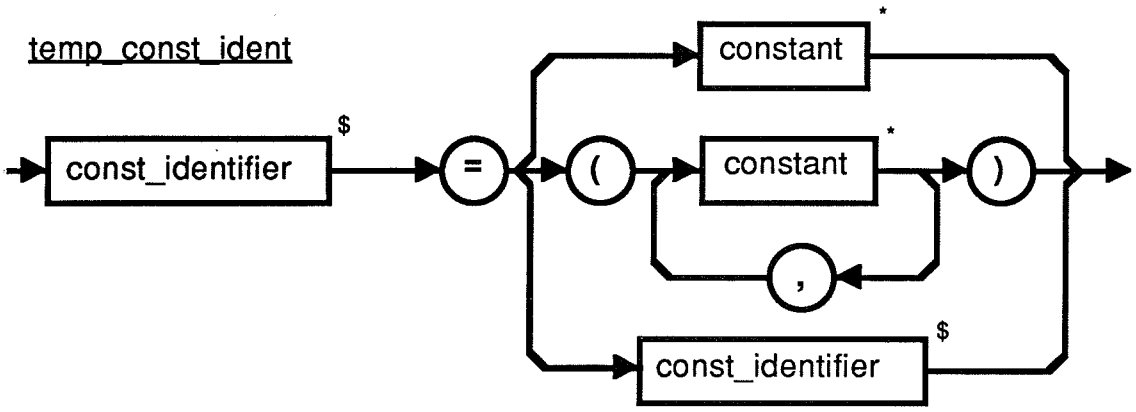
temp_const_spec



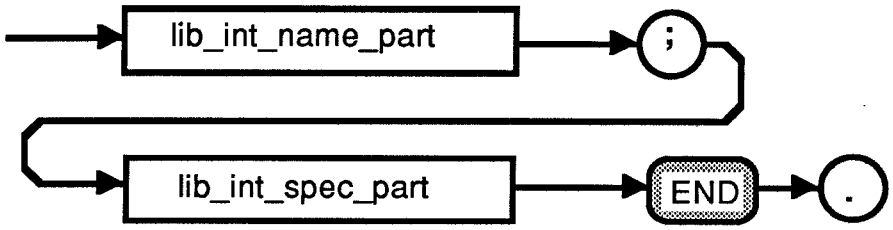
const_spec_list



temp_const_ident

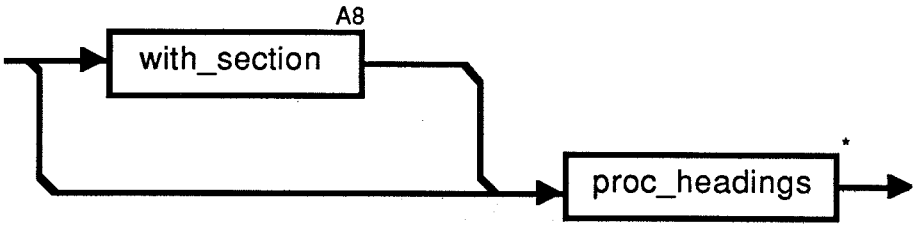


library_interface

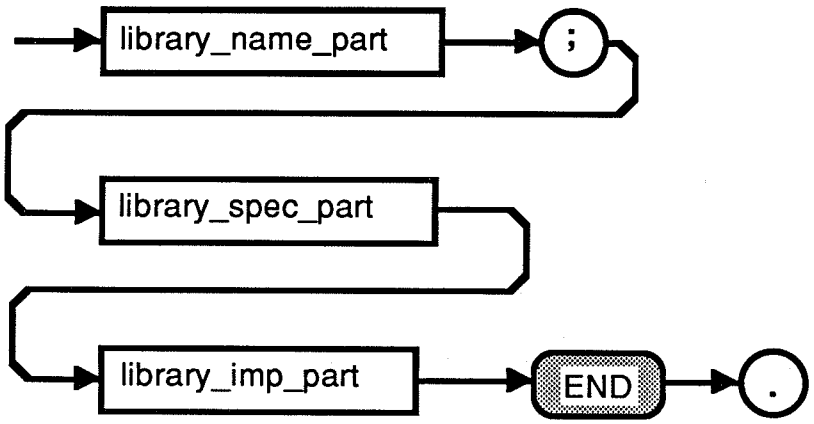


lib_int_name_part

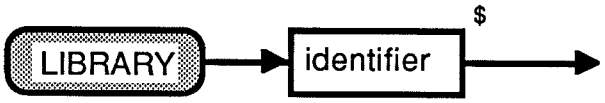
lib_int_spec_part



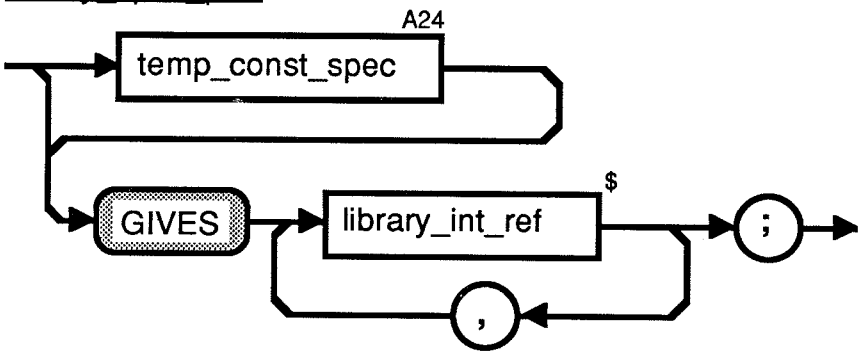
library



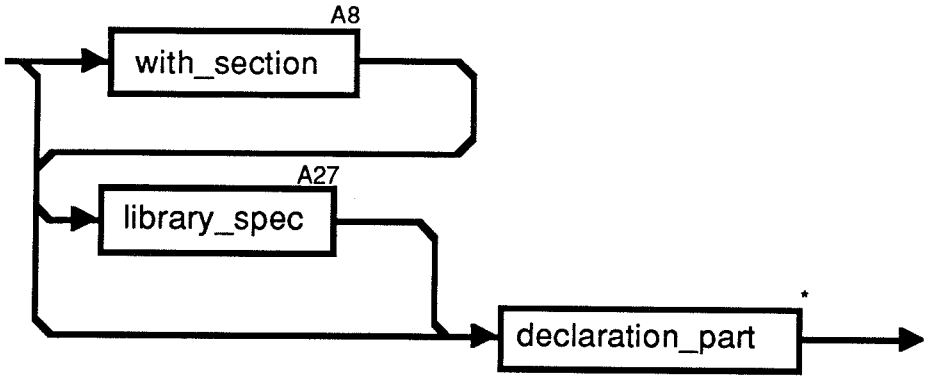
library_name_part



library_spec_part

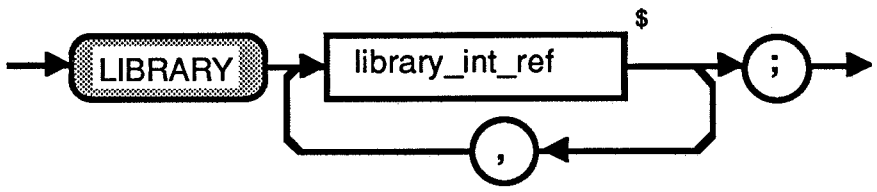


library_imp_part

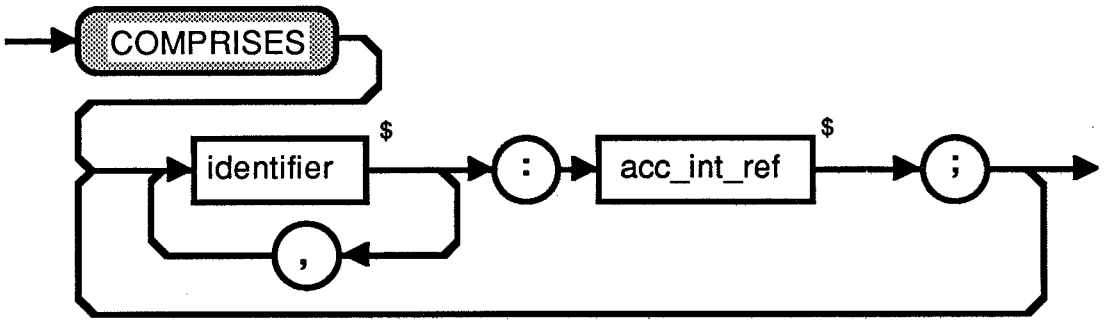


Variables may not be declared in a **library** implementation part.

library_spec



comp_acc_int_spec_part



BACHUS - NAUR FORM OF SYNTAX

The form of Bachus Naur syntax used is as follows :-

a) Normal font lower case words, some containing embedded underlines, are used to denote syntactic categories.

b) Bold font words and symbols are used to denote Mascot reserved words and symbols.

c) Underlined words are used to reference Pascal constructs. For further expansion of these constructs refer to a Pascal syntax definition.

d) Italicised prefix words are used to convey semantic qualification of the following non-italic root. For example, if an identifier is required which must be the name of a template, the following compound name is used:

template identifier

e) Square brackets enclose optional items. (NB. Bold square brackets occur in the syntax definition. These refer to symbols in the Mascot design language.)

f) Braces enclose a repeated item. The item may appear zero or more times.

g) A vertical bar separates alternatives. They are always used to separate alternatives for the construct being defined, never as alternatives for part of a construct.

Section 2.3

access_interface ::= access_interface_name_part **;**
    access_interface_specification_part **end .**

access_interface_name_part ::= **access interface** identifier

simple_access_interface_specification_part ::=
    [with_section] access_interface_detail_part

with_section ::=

with *definition*_identifier {, *definition*_identifier} ;

procedure_or_function_heading ::= procedure_heading | function_heading

definition_unit ::=

definition_name_part ; definition_specification_part **end .**

definition_name_part ::= **definition** identifier

definition_specification_part ::=

[with_section] definition_detail_part

definition_detail_part ::=

constant_definition_part

| [constant_definition_part] type_definition_part

port_specification ::= **requires** access_interface_declaration

{access_interface_declaration}

access_interface_declaration ::=

identifier_list : access_interface_definition ;

window_specification ::= **provides** access_interface_declaration

{access_interface_declaration}

Partially in Section 2.3, fully in Section 2.13

access_interface_definition ::= *access_interface*_identifier

| access_interface_array_description

Partially in Section 2.3, fully in Section 2.14

access_interface_specification_part ::=

simple_access_interface_specification_part

| composite_access_interface_specification_part

Partially in Section 2.3, fully in Section 2.15

access_interface_detail_part ::=
    [read_only_constant_specifications]
    | [read_only_constant_specifications] variable_specifications
    | [read_only_constant_specifications] [variable_specifications]
    procedure_or_function_heading {procedure_or_function_heading}

Section 2.4

subsystem ::= subsystem_name_part ; subsystem_specification_part
    network_implementation_part **end .**

subsystem_name_part ::= **subsystem** identifier

network_implementation_part ::=
    uses template_definition {, template_definition} ;
    component {component} [equivalence {; equivalence}]

component ::= component_class identifier : *template* identifier
    [connection_specification] ;

component_class ::= **activity | subsystem | server |**
    | ida | channel | pool | library

connection_specification ::= **(** connection {, connection} **)**

port_window_connect ::= *port* identifier **=** *component* identifier
    . *window* identifier

system ::= system_name_part ; system_specification_part
    system_implementation_part **end .**

system_name_part ::= **system** identifier

system_implementation_part ::= **uses** *template* identifier
    {, *template* identifier} ; component {component}

Partially in Section 2.4, fully in Section 2.8

subsystem_specification_part ::= [template_constant_definition]
[window_specification] [port_specification]

connection ::= template_constant_identity
| port_window_connect | port_port_connect

Partially in Section 2.4, fully in Section 2.14

template_definition ::= *composite_access_interface*_identifier
| *template*_identifier

port_port_connect ::= *port*_identifier = port_definition

port_definition ::= *boundary_port*_identifier
| *composite_port*_identifier . *comprises*_identifier

equivalence ::= window_window_equivalence | window_port_equivalence

window_window_equivalence ::= window_declaration = *component*_identifier
. *window*_identifier

window_port_equivalence ::= window_declaration = port_definition

window_declaration ::= *boundary_window*_identifier
| *composite_window*_identifier . *comprises*_identifier

Section 2.5

activity ::= activity_name_part ; activity_specification_part
activity_implementation_part **end .**

activity_name_part ::= **activity** identifier

Partially in Section 2.5, fully in Section 2.8

activity_specification_part ::= [template_constant_specification]
[port_specification]

Partially in Section 2.5, fully in Section 2.9

simple_activity_implementation_part ::= [with_section]
[library_specification] [declaration_part]
compound_statement

Partially in Section 2.5, fully in Section 2.12

activity_implementation_part ::= simple_activity_implementation_part
| composite_activity_implementation_part

Section 2.6

ida_name_part ::= ida_class identifier

ida_class ::= **channel** | **ida** | **pool**

access_equivalence ::= renaming_equivalence | simple_equivalence

renaming_equivalence ::=
*window*_identifier . *access_interface_declaration*_identifier
= renamed_declaration

renamed_declaration ::=
*internal_declaration*_identifier
| *port*_identifier . *port_declaration*_identifier

simple_equivalence ::= *window*_identifier = *port*_identifier

Partially in Section 2.6, fully in Section 2.8

ida_specification_part ::= [template_constant_specification]
window_specification [port_specification]

Partially in Section 2.6, fully in Section 2.9

ida ::= ida_name_part ; ida_specification_part
ida_implementation_part **end .**

simple_ida_implementation_part ::= [with_section]
[library_specification] declaration_part
[access_equivalence {; access_equivalence}]

Partially in Section 2.6, fully in Section 2.10

ida_implementation_part ::= simple_ida_implementation_part
| network_implementation_part

Section 2.7

server_name_part ::= **server** identifier

Partially in Section 2.7, fully in Section 2.9

server ::= server_name_part ; ida_specification_part
server_implementation_part **end .**

simple_server_implementation_part ::= [with_section]
[library_specification] declaration_part
[access_equivalence {; access_equivalence}]

server_implementation_part ::= simple_server_implementation_part
| network_implementation_part

Section 2.8

system_specification_part ::= [template_constant_specification]

template_constant_specification ::=
constant template_constant_group {template_constant_group}

template_constant_group ::= identifier_list :
[array [integer {, integer }] of] standard_scalar_type

template_constant_identity ::=
template_constant identifier = template_constant_definition

template_constant_definition ::=

(constant {, constant})

| constant

| *constant* identifier

Section 2.9

library_interface ::= library_interface_name_part ;

library_interface_specification_part **end .**

library_interface_name_part ::= **library interface** identifier

library_interface_specification_part ::= [with_section]

procedure_or_function_heading {procedure_or_function_heading}

library ::= library_name_part ; library_specification_part

library_implementation_part **end .**

library_name_part ::= **library** identifier

library_specification_part ::= [template_constant_specification]

gives *library_interface* identifier

{, *library_interface* identifier} ;

library_implementation_part ::= [with_section] [library_specification]

library_declarative_part

library_declarative_part ::= [constant_definition_part]

[type_definition_part] procedure_and_function_declaration_part

library_specification ::= **library** *library_interface* identifier

{, *library_interface* identifier} ;

Section 2.12

composite_activity_implemention_part ::=

uses template_definition {, *template* identifier} ;

activity_component_part {activity_component_part}

activity_component_part ::= activity_component_class identifier
: *template*_identifier [activity_connection_specification] ;

activity_component_class ::= **root** | **subroot** | **library**

activity_connection_specification ::= (activity_connection
{, activity_connection})

activity_connection ::= template_constant_identity
| sub_element_link | port_port_connect

sub_element_link ::= *out_link*_identifier = *subroot*_identifier

root ::= root_name_part ; root_specification_part
simple_activity_implementation_part **end** .

root_name_part ::= **root** identifier

root_specification_part ::= [activity_specification_part]
[needs_list]

needs_list ::= **needs** needed_interface {needed_interface}

needed_interface ::=
identifier_list : *subroot_interface*_identifier ;

subroot_interface ::= subroot_interface_name_part ;
simple_access_interface_specification_part **end** .

subroot_interface_name_part ::= **subroot interface** identifier

subroot ::= subroot_name_part ; subroot_specification_part
subroot_implementation_part **end** .

subroot_name_part ::= **subroot** identifier

subroot_specification_part ::= activity_specification_part
gives *subroot_interface*_identifier ; [needs_list]

subroot_implementation_part ::= simple_subroot_implementation_part
| composite_activity_implementation_part

simple_subroot_implementation_part ::= [with_section]
[library_specification] declaration_part

Section 2.13

access_interface_array_description ::=
array [simple_type {, simple_type}] **of**
*access_interface*_identifier

Section 2.14

composite_access_interface_specification_part ::=
comprises comprise_declaration {comprise_declaration}

comprise_declaration ::=
identifier_list : *access_interface*_identifier ;

Section 2.15

read_only_constant_specifications ::=
parameter_group {parameter_group}

variable_specifications ::=
var parameter_group {parameter_group}

Appendix B

mascot_3_unit ::= definition_unit | interface_unit | template_unit

interface_unit ::= access_interface | subroot_interface
| library_interface

template_unit ::= system | subsystem | activity | server | ida | root
| subroot | library

SYNTAX INDEX

In the list given below, each syntactic category is followed by the section where it is defined. In addition, each syntactic category is followed by the names of other categories in whose definition it appears. An ellipsis (...) is used when the syntactic category is or can be a reserved word or symbol. All uses of parentheses are combined in the form "()". The italicised prefixes used with some terms are deleted here.

renamed_declaration	2.6
renaming_equivalence	2.6
root_name_part	2.12
server_name_part	2.7
simple_equivalence	2.6
sub_element_link	2.12
subroot_interface_name_part	2.12
subroot_name_part	2.12
subroot_specification_part	2.12
subsystem_name_part	2.4
system_implementation_part	2.4
system_name_part	2.4
template_constant_definition	2.8
template_constant_identity	2.8
template_definition	2.4, 2.14
window_declaration	2.4, 2.14
window_window_equivalence	2.4, 2.14
with_section	2.3
identifier_list	Pascal
access_interface_declaration	2.3
comprise_declaration	2.14
needed_interface	2.12
template_constant_group	2.8
integer	...
template_constant_group	2.8
interface	...
access_interface_name_part	2.3
library_interface_name_part	2.9
subroot_interface_name_part	2.12
interface_unit	Appendix B
mascot_3_unit	Appendix B

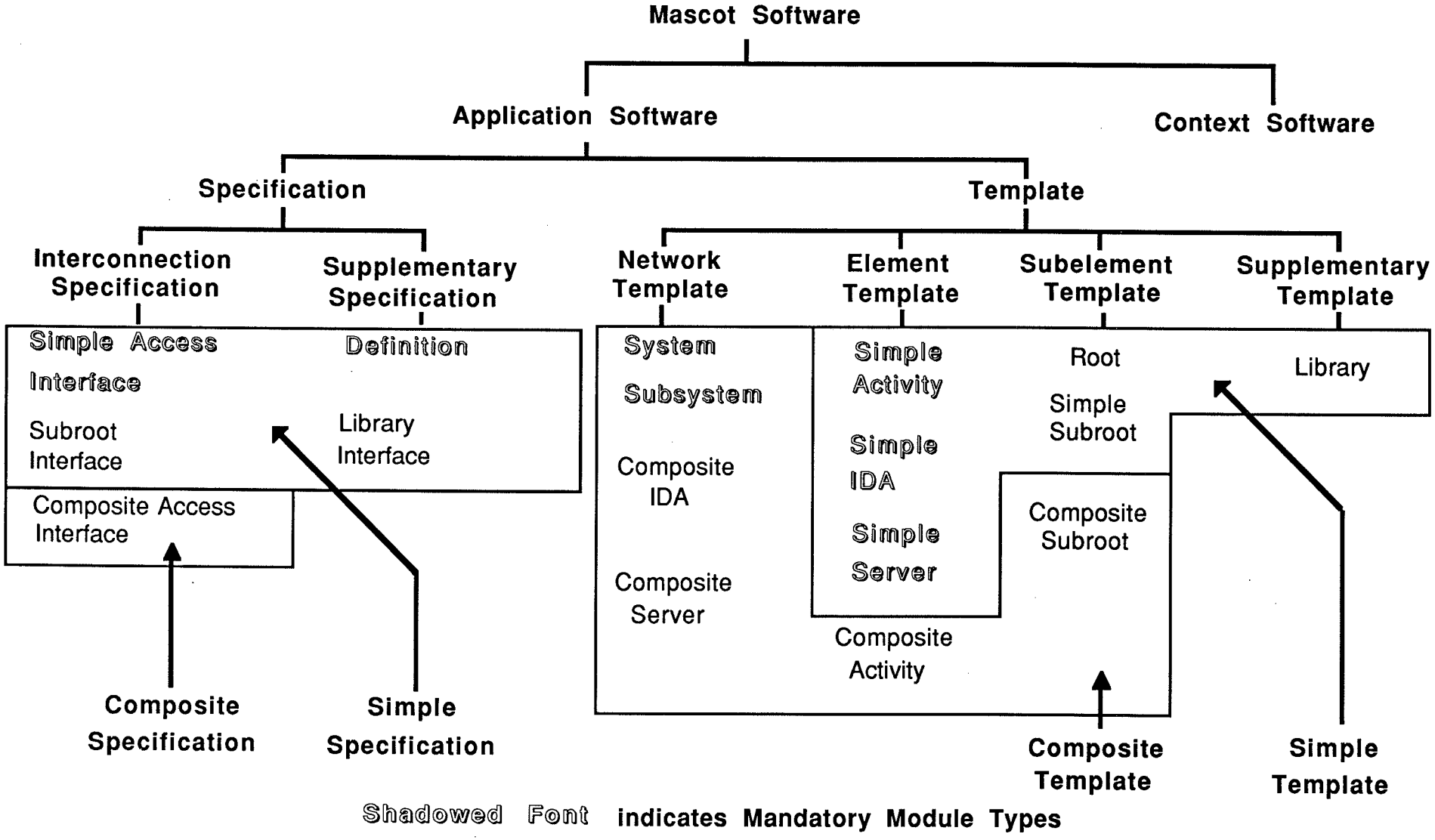
template_constant_definition	2.8
subsystem_specification_part	2.4, 2.8
template_constant_identity	2.8
template_constant_identity	2.8
activity_connection	2.12
connection	2.4, 2.8
template_constant_group	2.8
template_constant_specification	2.8
template_constant_specification	2.8
activity_specification_part	2.5, 2.8
ida_specification_part	2.6, 2.8
library_specification_part	2.9
system_specification_part	2.8
template_definition	2.4, 2.14
composite_activity_implementation_part	2.12
network_implementation_part	2.4
template_unit	Appendix B
mascot_3_unit	Appendix B
type_definition_part	Pascal
definition_detail_part	2.3
library_declarative_part	2.9
uses	...
composite_activity_implementation_part	2.12
network_implementation_part	2.4
system_implementation_part	2.4
var	...
variable_specifications	2.15
variable_specifications	2.15
access_interface_detail_part	2.3, 2.15

template_constant_definition	2.8
template_constant_group	2.8
with_section	2.3
	...
access_interface	2.3
activity	2.5
definition_unit	2.3
ida	2.6, 2.9
library	2.9
library_interface	2.9
port_definition	2.4, 2.14
port_window_connect	2.4
renamed_declaration	2.6
renaming_equivalence	2.6
root	2.12
server	2.7, 2.9
subroot	2.12
subroot_interface	2.12
subsystem	2.4
system	2.4
window_declaration	2.4, 2.14
window_window_equivalence	2.4, 2.14
	...
activity_component_part	2.12
access_interface_declaration	2.3
component	2.4
comprise_declaration	2.14
needed_interface	2.12
template_constant_group	2.8
	...
access_interface	2.3
access_interface_declaration	2.3
activity	2.5
activity_component_part	2.12
component	2.4
composite_activity_implementation_part	2.12
comprise_declaration	2.14
definition_unit	2.3

	ida	2.6, 2.9
	library	2.9
	library_interface	2.9
	library_specification	2.9
	library_specification_part	2.9
	needed_interface	2.12
	network_implementation_part	2.4
	root	2.12
	server	2.6, 2.9
	simple_ida_implementation_part	2.6, 2.9
	simple_server_implementation_part	2.7, 2.9
	subroot	2.12
	subroot_interface	2.12
	subroot_specification_part	2.12
	subsystem	2.4
	system	2.4
	system_implementation_part	2.4
	with_section	2.3
=		...
	port_port_connect	2.4, 2.14
	port_window_connect	2.4
	renaming_equivalence	2.6
	simple_equivalence	2.6
	sub_element_link	2.12
	template_constant_identity	2.8
	window_port_equivalence	2.4, 2.14
	window_window_equivalence	2.4, 2.14
[ ]		...
	access_interface_array_description	2.13
	template_constant_group	2.8

APPENDIX B

MODULE TAXONOMY


Mascot Software
Application Software
Context Software
Specification
Template
Interconnection Specification
Supplementary Specification
Network Template
Element Template
Subelement Template
Supplementary Template
Simple Access Interface
Definition
Subroot Interface
Library Interface
Composite Access Interface
System Subsystem
Composite IDA
Composite Server
Simple Activity
Simple IDA
Simple Server
Composite Activity
Root
Simple Subroot
Composite Subroot
Library
Composite Specification
Simple Specification
Composite Template
Simple Template
Shadowed Font indicates Mandatory Module Types

APPENDIX C

SUMMARY OF KEYWORD USAGE

USE OF KEYWORDS

		CONSTANT	PROVIDES	REQUIRES	GIVES	NEEDS	COMPRISES	WITH	USES	LIBRARY
Specification Modules	Definition	-	-	-	-	-	-	0+	-	-
	Simple Access Interface	-	-	-	-	-	-	0+	-	-
	Subroot Interface	-	-	-	-	-	-	0+	-	-
	Library Interface	-	-	-	-	-	-	0+	-	-
	Composite Access Inteface	-	-	-	-	-	1+	-	-	-

		Specification Dependencies						Implementation Dependencies		
Template Modules	Simple IDA *	0+	1+	0+	-	-	-	0+	-	0+
	Simple Activity	0+	-	0+	-	-	-	0+	-	0+
	Simple Root	0+	-	0+	-	0+	-	0+	-	0+
	Simple Subroot	0+	-	0+	1	0+	-	0+	-	0+
	Library	0+	-	-	1+	-	-	0+	-	0+
	Composite IDA *	0+	1+	0+	-	-	-	-	1+	0+
	Composite Activity	0+	-	0+	-	-	-	-	1+	0+
	Composite Subroot	0+	-	0+	1	0+	-	-	1+	0+
	Subsystem	0+	0+	0+	-	-	-	-	1+	0+
	System	0+	-	-	-	-	-	-	1+	0+

- = Prohibited 1 = One and only one

1+ = One or More 0+ = Zero or more

* Channel, Pool and Server have identical characteristics

APPENDIX D

DEFINITION OF GRAPHIC CONVENTIONS

DEFINITION OF GRAPHICAL CONVENTIONS

The purpose of documentation is to convey information to the reader. One technique for achieving this is the use of diagrams. The Mascot definition contains equivalent graphical and textual design representations. Conventions are defined for annotating the diagrams in the graphical representation so that they reflect all but the program code of the corresponding **modules**. However, a diagram containing too much detail can actually convey less information than a less detailed diagram. The optimum level of detail could well depend on the purpose for which the diagram is intended. Design and implementation documents, for example, form two relatively independent sets. Documents may be explanatory or definitive in purpose. In the course of design, omission of detail may reflect the postponment of decisions.

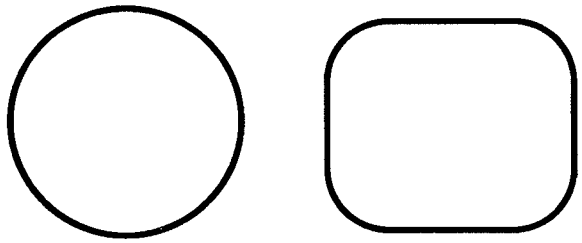
The definition recognises sufficient variability in the use of the standard graphic conventions to allow for local variance and for the desirability of employing different levels of detail for different purposes provided that consistency is maintained within sets of related documents.

Symbology

Symbols are introduced, in the appropriate sections of the Handbook, for the various entities employed in Mascot designs. This appendix presents the complete set of symbols and is to be regarded as the definitive document for this purpose.

Simple Forms

The **simple** forms of **activities**, **channels**, **pools**, generalised **IDAs**, **servers** and **subroots**, together with **roots**, constitute the **elements** and **subelements** of a Mascot design (see Appendix B). They are symbolised as follows:

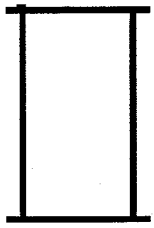


Simple Activity

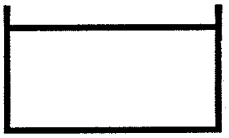
Root

Simple Subroot

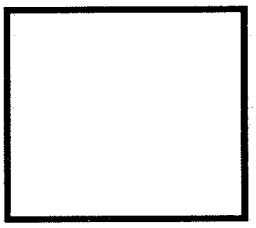
In accordance with long established Mascot practice, the circular form should normally be used. The rounded rectangular form is appropriate where it is desirable to provide additional space for the presentation of information within the boundary of the symbol. This might be the case, for example, with an **activity**, **root** or **subroot** which possesses an unusually large number of network and/or **subelement** connections.



Channel

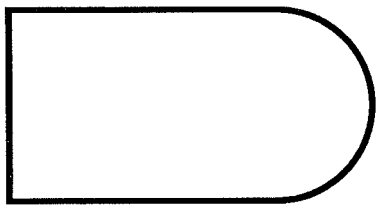


Pool



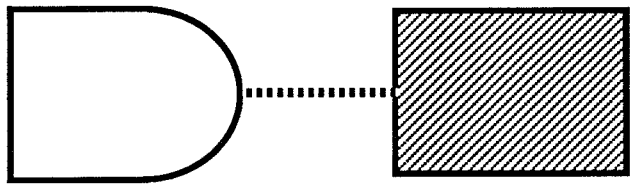
Simple IDA

The **channel** and **pool** symbols are based on those used in earlier versions of Mascot. These have been modified so that, like the **activity** symbols, they allow information to be presented within the boundary of the symbol. They may be used where an **IDA** conforms to the accepted definition (see Section 2.6) of one of these special forms. The rectangular symbol is a generalised form of **IDA**. It may be used to represent any **IDA** but must be employed where the **IDA** in question is not strictly a **channel** or a **pool**.



Simple Server

The symbol which represents a **server** is a combination of those representing **activities** and **IDAs**. It thus emphasises the hybrid nature of the **server** which behaves in the passive manner of an **IDA** as far as network interactions are concerned but, by virtue of being permitted to contain interrupt **handlers**, also possesses an active aspect. If the device to which the **server** is connected is also to be depicted on the diagram, it should be in the form shown below:



Device

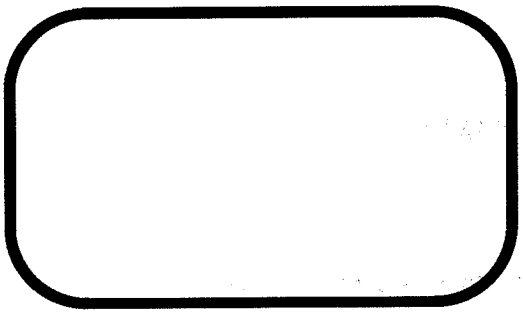
(with connection

to server)

The hatched rectangle may, however, may be replaced by a schematic drawing of the device.

Composite Forms

The same symbol is used to represent a **template** and a **component**. By definition, a **component** must be part of a **composite template**. This may be a **system** or a **subsystem**:

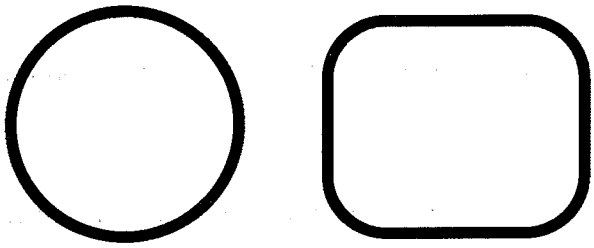


System

Subsystem

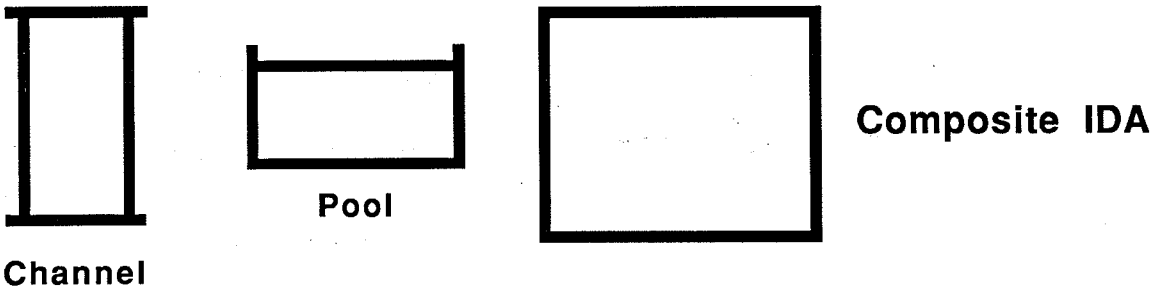
which is symbolised by any smooth closed curve, drawn with a thicker line than that used for the **simple** entities, but should normally take the form of a rounded rectangle.

For the **composite** forms of **activity**, **IDA**, **server** or **subroot**, the symbols used are the same as for the **simple** forms but normally drawn with thicker lines as they are throughout the Handbook.



Composite Activity

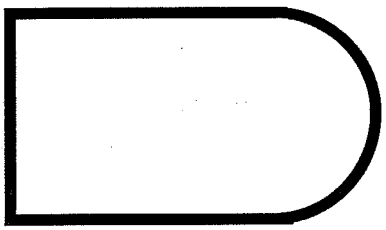
Composite Subroot



Composite IDA

Pool

Channel

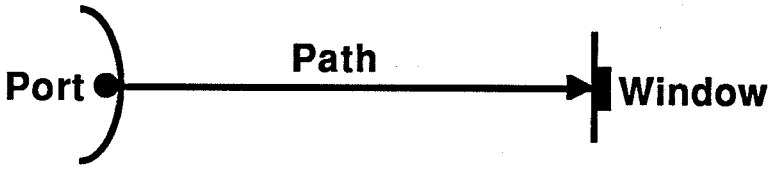


Composite Server

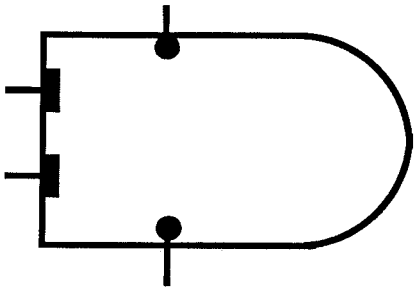
Alternatively, for all the **composite** forms, double lines, or some other convenient locally defined means of making the distinction, may be used. The same convention should be employed to denote **composite components** where the information is known and is considered relevant.

Paths, Ports and Windows

Symbols, illustrated below, are defined for **simple ports** and **windows** and the **path** which joins them. The lines which denote data **paths** may, if desired, be drawn as curves. Arrowheads are used to denote the direction of data flow and should normally be shown.

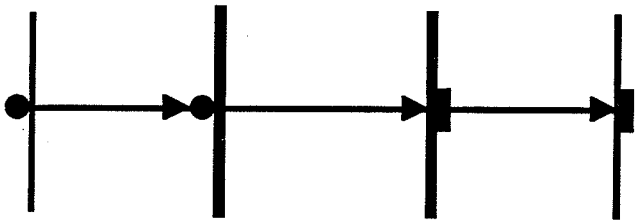


There are special rules governing the placement of **ports** and **windows** in **servers**:



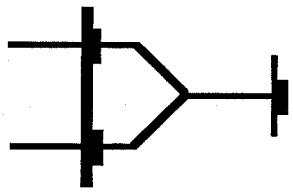
The **windows** should appear on the straight edge opposite to the semi-circular end of the symbol and the **ports** on either of the two adjacent sides.

Where **paths** pass through the boundaries of enclosing **composite** design entities, **port** or **window** symbols and data flow arrowheads are repeated at each boundary:



Path through Constructional Boundaries

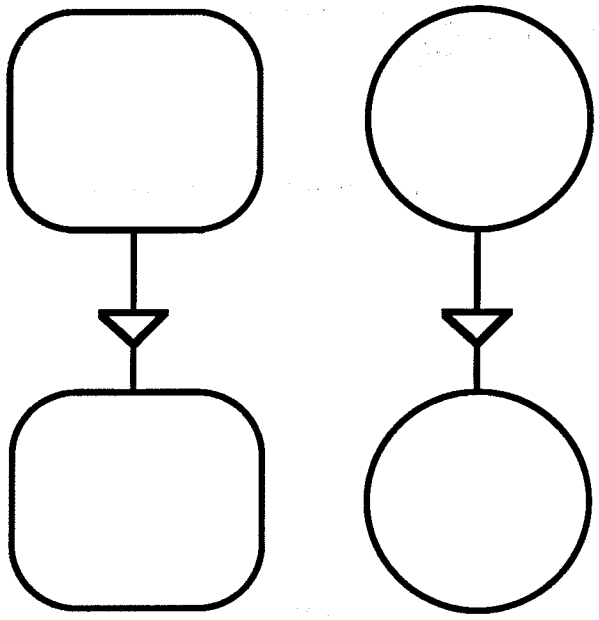
Since any number of **ports**, of the appropriate type, may be connected to a single **window**, diagrams frequently depict **paths** which merge. Normally, these lines are joined at a **window**. However, where it is more convenient, they may be merged at an intermediate point provided that merging is in the direction from **port** to **window**. This is illustrated in the diagram below in which the direction of data flow has deliberately been omitted as irrelevant to the point at issue.



Merging Paths

Subelements and Subelement Links

The **subelement links**, between the **components** of a **composite activity**, are represented graphically as follows:

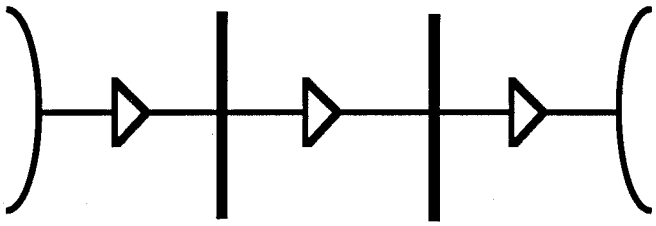


Root

Subelement Link

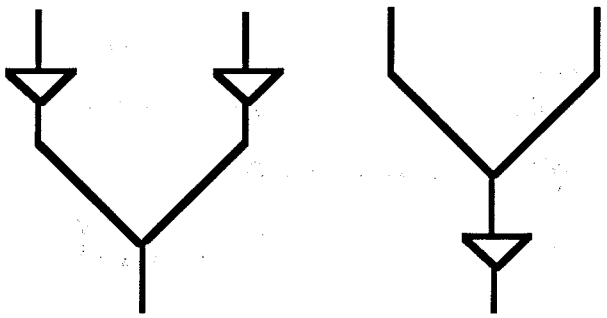
Simple Subroot

where the hollow arrowheads indicate the direction of procedure invocation and may be repeated where the **link** crosses enclosing boundaries:



Subroot Link through Constructional Boundaries

Links may be merged, in a manner similar to that employed for **paths**, in the direction of procedure application:



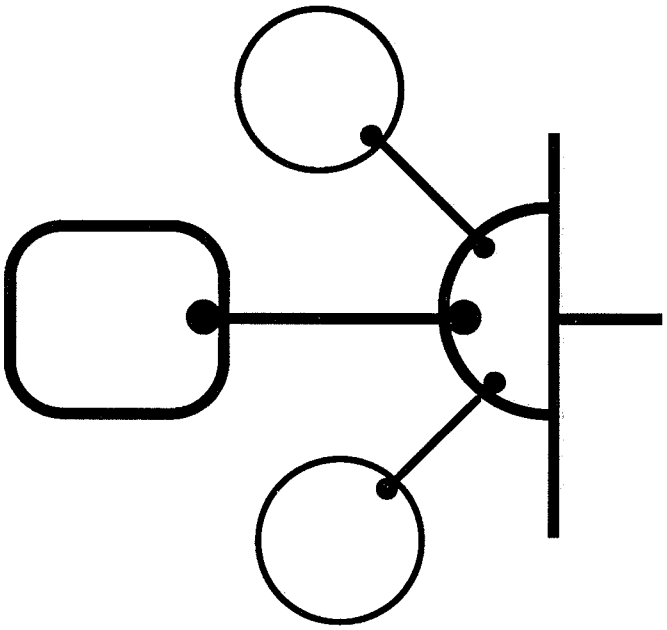
Merging

Subelement Links

Composit Paths, Ports and Windows

For the **template** in which a **composite access interface** is decomposed into its constituent **interfaces**, special symbols are defined to denote the **composite ports** and **windows**. At this level of decomposition, the **composite path** should be denoted by a thick (or double) line, where this is considered relevant:

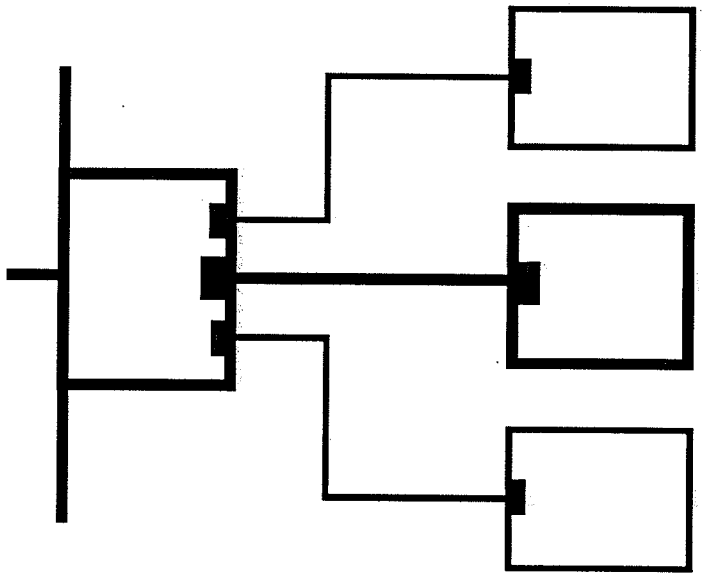
The **port** and **window** at each end of a **composite path** may be represented by a proportionately larger than normal **port** or **window** symbol if so desired.



Composite Port

The internal structure of a **composite port** is represented by a semi-circle, drawn with a thick line, and with its component **ports** illustrated inside the boundary.

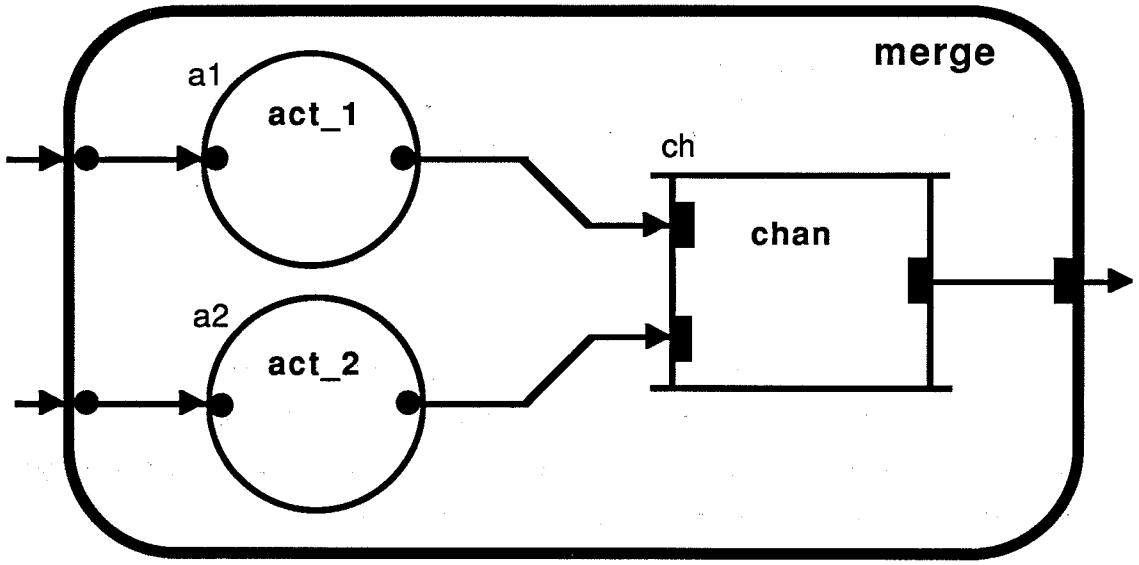
Composite Window



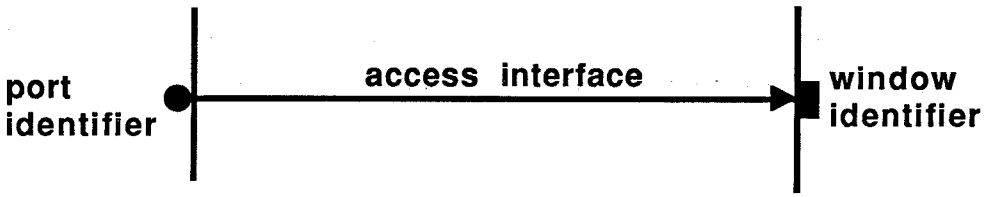
The internal structure of a **composite window** is represented by a rectangle, drawn with a thick line, and with its component **windows** illustrated inside the boundary.

Annotation

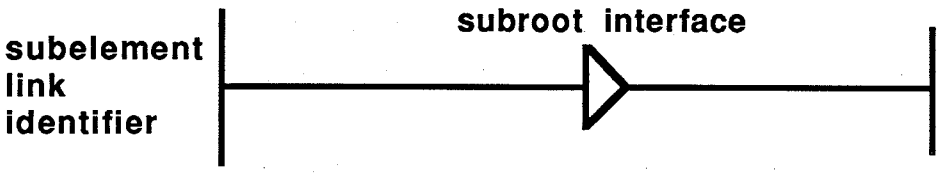
A **template** is annotated inside the symbol by the **template** name. A **component** of a **network** is annotated inside the symbol by the corresponding **template** name and on the outside by the **component** name. The example given below shows a **subsystem template** called ***merge*** . Its **components** are called ***a1*** , ***a2*** and ***ch*** and are derived from the **templates** ***act_1*** , ***act_2*** and ***chan*** , respectively.



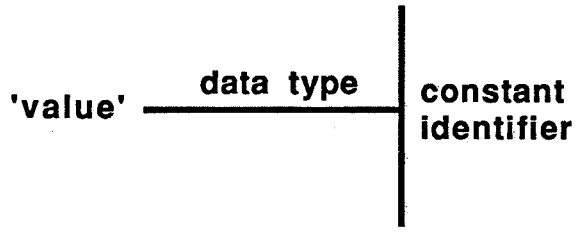
A **path** is annotated by the name of the corresponding **access interface** and **windows** and **ports** by their local names:



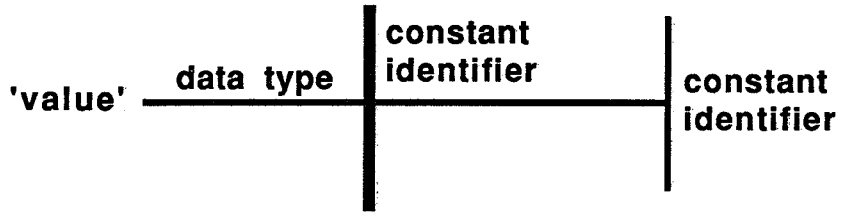
A **subelement link** is annotated with the name of the corresponding **subroot interface** and the active end of the **link** by its local name:



Where a **component** or **template** possesses a **template constant**, the name, type and value of the constant may be shown on the diagram:



and the value may be depicted as being supplied across one or more enclosing boundaries:



Annotation Options

In the sense of their inclusion in a Mascot diagram, **template** names and **path** names are essential but **component** names and **window/port** names are less important. This is especially so where no ambiguity arises as, for example, where the **components** are all derived from different **templates** or the **windows** or **ports** of a **template** or **component** are of different (**access interface**) types.

Extensions

In addition to the name associated with an **access interface**, it may be considered desirable to annotate a **path** by the name of the data objects and/or the type of the data objects that flow along the **path**. This could be indicated by the name of the **definition module** which defines the type of the object.

Port, **window** and **interface** qualifiers can also be used to annotate the appropriate features of a diagram.

Although intended to denote only one level of decomposition at a time, it is sometimes useful to show multiple levels of decomposition in one diagram. The consequent increase in the amount of information to be conveyed would probably necessitate showing only a subset of that available.

The Mascot diagram can give a one-to-one representation of the information contained in the **specification part** of a **simple module**. It is therefore biased towards design expression. However, during design derivation, it may be highly desirable to use 'Mascot-like' diagrams which use a subset of the standard conventions. For example, a **path** may be identified between two **components** before the decision is made as to which **component** possesses the **port** and which the **window** (which possesses the 'motive power'). This is acceptable.

For information purposes only, it may be desirable to generate Mascot diagrams showing only a subset of the **components** (for example omitting **pools** or **channels** or **servers**). Again this is acceptable provided the diagrams are described as such.

APPENDIX E

CLASSIFIED SUMMARY OF MASCOT FEATURES

CLASSIFIED SUMMARY OF MASCOT FEATURES

Introduction

The purpose of this appendix is to provide a summary of all the Mascot features defined elsewhere in the Handbook and to identify those parts of the definition which are mandatory in any Mascot 3 development environment. These mandatory features have been selected on the basis of a standardisation philosophy whose objectives are to enable products to be assessed for conformance with the definition and to enable designs to be portable between Mascot development environments. This philosophy is presented below.

Philosophy of Standardisation

As the first element in the philosophy of standardisation, every effort has been made to define all the Mascot features in an unambiguous manner. Secondly, a mandatory subset of the definition has been identified. This is reflected in the order of presentation of material in the Handbook sections on the design representation facilities and is summarised for reference at the end of this appendix. All mandatory features must be provided by any Mascot 3 development environment. Where features not included in this subset are implemented, they are required to conform to the definition given in the Handbook.

The third element of the philosophy is the requirement that a development environment include adequate linguistic support for the features which it implements. The design representation language used in the Handbook provides definitive guidance as to what concepts need to be expressed but need not necessarily be literally implemented. The linguistic support may take the form of an Additional Features design language (cf AF Coral 2 for Mascot 2) for the programming language or languages supported by the development environment. This should be considered the preferred option as it will reduce the resources required to verify design compliance of Mascot software and will enhance software portability.

Alternatively, the Mascot 3 module classes could be implemented in a native programming language, without additional features, by a code of practice. In this case the mapping of each characteristic of each module class onto the programming language must be defined as part of the code of practice. This should be done in a form which provides a suitable basis for validation testing, for transporting a design and for design conformance checking or other verification of application software designs.

The fourth element of the philosophy is that module classes do not need to be supported in full. A mandatory subset of characteristics has been defined for each class. Thus the definition contains characteristics which not all development environments will implement even within the mandatory classes. This element of the philosophy guarantees a minimum level of portability as it defines the

minimum characteristics of all Mascot 3 development environments. It also defines the minimum set of characteristics which must be provided for each non mandatory module class implemented. Where a non mandatory characteristic is supported it must be supported in all appropriate module classes. It is recognised that design authorities may wish to control or restrict the use of certain facilities, such as direct data visibility, and therefore warning reports of their use may be generated by a development environment.

The fifth and final element of the philosophy concerns extensions to Mascot 3. It is recognised that the definition cannot be maintained, either in scope or in functionality, in advance of users' requirements and that therefore certain applications will need facilities added to their development environments that are not covered in the Mascot 3 definition. Implementing a superset of the facilities, whilst not encouraged, is therefore permitted. Implementors must define fully any extensions provided and must declare them to any testing authority so that they may be documented in a test report. Such additional features must not, of course, interfere with or partially overlap any of the defined facilities.

The Mandatory Subset

The mandatory facilities have been divided into three categories: module classes, commands and primitives. The mandatory facilities specified below must be provided by all Mascot 3 development environments. However, users are not obliged to use all the mandatory features. Thus, for example, all development environments must support the use of the WITH keyword within **access interfaces**. However, it is possible to have a syntactically correct **access interface** which does not use the feature. It is not required that support for unused features be present in target systems.

(a) Module Classes

The diagram of Appendix B shows the complete set of Mascot module classes, the mandatory subset being highlighted by means of a shadowed font. All Mascot 3 development environments must provide these classes. The first of the tables below specifies all the defined characteristics of the subset and identifies, with an **M** for mandatory, those features which it is obligatory to provide.

The second table specifies all the features of the non-mandatory module classes. Mascot 3 development environments which support any of these classes must provide support for use of the characteristics identified with an **M**.

The third table specifies the features of particular module classes which become mandatory when specific non mandatory features are supported.

(b) Commands

It is mandatory to provide facilities for registration, introduction, enrolment, building, initialisation and starting which accord with the Mascot definition. They may be provided in one of two ways: either

explicitly through a command interpreter in the form shown in the fourth table below, or by use of other user facilities such as those provided by most computer operating systems. When a command interpreter is not being used the documentation of the Mascot development environment must define explictly how the functions are provided.

(c) Primitives

There are no mandatory primitives in Mascot 3. However, there are inter-dependencies between the primitives described in the Handbook. The fifth of the tables below divides these primitives into sets which are required to be implemented together as a group. It also indicates which of the other primitive groups are also mandatory for each group supported.

1. CHARACTERISTICS OF MANDATORY MODULE CLASSES

MODULE CLASS		CHARACTERISTIC
ACCESS INTERFACE	**M**	**WITH**
	M	Procedure specifications
		Variable specifications
		Read-only data constants
		Qualifiers
DEFINITION		**WITH**
		Symbolic constants
	M	Type definitions
IDA		**CONSTANT**
	M	**PROVIDES**
	M	**REQUIRES**
		WITH
		LIBRARY
		Window qualifiers
		Port qualifiers
	M	Data area
	M	Access procedures
		Access data
	M	Window-to-local equivalence
		Window-to-remote equivalence
		Window-to-port equivalence
		Arrays of **CONSTANT**s
		Arrays of ports
		Arrays of windows
	M	Initialisation procedure
		Reset procedure
		Termination procedure
SERVER		**CONSTANT**
	M	**PROVIDES**
	M	**REQUIRES**
		WITH
		LIBRARY
		Window qualifiers
		Port qualifiers
	M	Data area
		Handler
	M	Access procedures
		Access data
	M	Window-to-local equivalence
		Window-to-remote equivalence
		Window-to-port equivalence
		Handler-to-interrupt connection
		Arrays of **CONSTANT**s
		Arrays of ports
		Arrays of windows
	M	Initialisation procedure
		Reset procedure
		Termination procedure

MODULE CLASS		CHARACTERISTIC
ACTIVITY		**CONSTANT**
	M	**REQUIRES**
		WITH
		LIBRARY
		Port qualifiers
		Arrays of **CONSTANT**s
		Arrays of ports
SUBSYSTEM		**CONSTANT**
	M	**PROVIDES**
	M	**REQUIRES**
	M	**USES**
		Window qualifiers
		Port qualifiers
		Arrays of **CONSTANT**s
		Arrays of ports
		Arrays of windows
		Library instantiation
	M	IDA components
	M	SERVER components
	M	ACTIVITY components
	M	SUBSYSTEM components
	M	Window-to -window equivalence
		Window-to-port equivalence
SYSTEM		**CONSTANT**
	M	**USES**
		Library instantiation
	M	IDA components
	M	SERVER components
	M	ACTIVITY components
	M	SUBSYSTEM components
		Arrays of **CONSTANT**s

2. CHARACTERISTICS OF NON-MANDATORY MODULE CLASSES

MODULE TYPE		CHARACTERISTIC
COMPOSITE ACCESS INTERFACE	M	COMPRISES
SUBROOT INTERFACE	M	WITH
	M	Procedure specifications
		Variable specifications
		Read-only data constants
LIBRARY INTERFACE	M	WITH
		Procedure specifications
		Read-only data constants
COMPOSITE IDA		CONSTANT
	M	PROVIDES
	M	REQUIRES
	M	USES
		Window qualifiers
		Port qualifiers
		Arrays of CONSTANTs
		Arrays of ports
		Arrays of windows
		Library instantiation
	M	IDA components
	M	Window-to-window equivalence
		Window-to-port equivalence
COMPOSITE ACTIVITY		CONSTANT
	M	REQUIRES
	M	USES
		Port qualifiers
		Arrays of CONSTANTs
		Arrays of ports
		Library instantiation
	M	ROOT components
	M	SUBROOT components
ROOT		CONSTANT
	M	REQUIRES
	M	NEEDS
		WITH
		LIBRARY
		Port qualifiers
		Arrays of CONSTANTs
		Arrays of ports

MODULE CLASS		CHARACTERISTIC
SUBROOT		**CONSTANT**
	M	**REQUIRES**
	M	**GIVES**
	M	**NEEDS**
		WITH
		LIBRARY
		Port qualifiers
		Arrays of **CONSTANT**s
		Arrays of ports
COMPOSITE SUBROOT		**CONSTANT**
	M	**REQUIRES**
	M	**GIVES**
	M	**NEEDS**
	M	**USES**
		Port qualifiers
		Arrays of **CONSTANT**s
		Arrays of ports
	M	SUBROOT components
		Library Instantiation
LIBRARY		**CONSTANT**
		GIVES
		WITH
		LIBRARY
		Arrays of **CONSTANT**s
COMPOSITE SERVER		**CONSTANT**
	M	**PROVIDES**
	M	**REQUIRES**
	M	**USES**
		Window qualifiers
		Port qualifiers
		Arrays of **CONSTANT**s
		Arrays of ports
		Arrays of windows
		Library instantiation
	M	IDA components
	M	SERVER components
	M	Window-to-window equivalence
		Window-to-port equivalence

3. MANDATORY GROUPS OF MASCOT 3 MODULE CLASSES

Group Name	Group	Requirements
COMPOSITE ACTIVITY	**COMPOSITE ACTIVITY** **SUBROOT INTERFACE** **ROOT** **SUBROOT**	Implement as a group
COMPOSITE SUBROOT	**COMPOSITE SUBROOT**	Implement COMPOSITE ACTIVITY group
LIBRARY	**LIBRARY INTERFACE** **LIBRARY** template	Support keyword **LIBRARY** Support library instantiation at least in **SYSTEM**

4. MANDATORY GROUPS OF MASCOT 3 OPERATIONS

Group Name	Group
Status progression	**REGISTER** **INTRODUCE** **ENROL**
Building	**BUILD**
Execution Control	**INITIALISE** **START**

5. GROUPS OF MASCOT 3 PRIMITIVES

Group Name	Group	Other Groups Required
Control Queue	**JOIN** **LEAVE** **WAIT** **STIM**	
Checking	**CHECK**	Control Queue
Basic Interrupt	**STIMINT** **ENDHANDLER**	Control Queue
Interrupt Connection	**CONNECT**	Basic Interrupt
Interrupt Disconnection	**DISCONNECT**	Interrupt Connection
Timing	**TIMENOW** **DELAY**	
Timeout	**WAITFOR**	Control Queue Timing
Co-operative Scheduling	**SUSPEND**	
Activity Termination	**ENDROOT**	
Error Handling	**GETERROR** **ERROR** **FATAL_ERROR**	
Monitoring	**SELECT** **EXCLUDE** **RECORD**	
Execution Control	**HALT** **RESUME** **START** **TERMINATE** **RESET**	

If any primitive in a group is implemented then all primitives in that group and in any 'groups required' must be implemented.

APPENDIX F

GLOSSARY

MASCOT GLOSSARY

Access Interface A **specification** defining the possible interactions (eg procedure specifications) between the **components** connected by a **path**. In its **composite** form it comprises a set of other **access interfaces**.

Access Procedure A procedure, implemented in an **IDA** or **server**, and corresponding to a procedure heading specified in an **access interface** to which a **window** specification of the **IDA** or **server** refers. It provides a network interaction along a connected **path** of the appropriate type.

Activity The Mascot design entity representing a single independent information processing element. An activity **module** is a **template** which may be used to create activity **components** each of which is an independently scheduled single sequential program thread, conceptually executing in parallel with other activities. An activity usually specifies one or more **ports** each of which defines a connection to be established from the activity to a **window** on a neighbouring **component** in the execution environment. A **composite** form of activity is provided for sequential program decomposition in terms of **subelements** known as **roots** and **subroots.**

Building The process by which executable software is created from its defining **templates**, which must have achieved **fully enrolled status**.

Channel A special case of an **IDA** having destructive read properties. The channel provides facilities for transmission of information.

Class The category of a Mascot design entity. Design entities are grouped into **specifications** and **templates**. **Specifications** comprise the classes: **access interface**, **subroot interface**, **library interface** and **definition**. **Templates** comprise the classes: **system**, **subsystem**, **activity**, **channel**, **pool**, **IDA**, **server**, **root**, **subroot** and **library**.

Component A constituent of a **composite template**. The type of a component is identified by the name of the **template** which is used in the definition of the component.

Composite A composite **module** is one which is further decomposed in terms of lower level **modules**. For **specifications** the decomposition is in terms of the same **class** of **specification**. For **templates** the decomposition depends on the **class** of the **template**. The only **specification** with a composite form is the **access interface**. The following **templates** have composite forms: **system, subsystem, IDA, server, activity, subroot**.

Context That part of the executable software which supports (but is not part of) the application software defined by the **system template**. It implements the facilities specified in the **context interface**.

Context Interface A special form of **specification** which defines the facilities offered (implicitly) by the **context** to all applications **modules**. It is usually a collection of other **specifications**.

Control Queue An object, declared in an **IDA**, which may be operated upon by a set of primitives to ensure mutual exclusion and cross-stimulation between **activities**. The relevant primitives are CHECK, JOIN, LEAVE, STIM, STIMINT, WAIT, WAITFOR.

Definition A **specification** defining a set of data types and named constants for use by **simple interfaces** and **simple templates**. It may refer to other definition **modules**.

Element A fundamental Mascot design entity: **simple activity, simple IDA, simple server, composite activity**.

Enrol The enrol operation checks that the **name part**, **specification part** and **implementation part** of a **template module** have been defined and are legal. It may involve using information in the **specification parts** of the **modules** to which it refers. If the checking is successful, then the **module** will be accorded **partially** or **fully enrolled status**, depending upon the type of the **module** itself and on the **status** of the **modules** to which it refers.

Fully Enrolled A **status** value achieved by a **template module** as a result of a successful **enrol** operation. For a **composite template**, the **status** indicates that all the **templates** which define the **module's components** have also achieved **fully enrolled status**.

Fully Introduced A **status** value achieved by a **module** as the result of a successful **introduce** operation. The **status** indicates that all **specification modules** referred to in the **module's specification part** have also achieved **fully introduced status**.

Handler A routine invoked as a direct consequence of a hardware interrupt. All handlers are located within **servers**.

IDA The Mascot design entity representing a passive independent information storage or information transmission element. In the **simple** form it contains both shareable data and the access mechanisms which can operate on this data; these access mechanisms safeguard the integrity of information within or passing through an IDA and sustain information propagation for intercommunication between **activities**. An IDA **module** is a **template** which may be used to create IDA **components** each of which allows several independently scheduled single sequential program threads to be simultaneously active or suspended within the IDA. An IDA specifies one or more **windows** defining the connections which can be established from **ports** on neighbouring **components** in the execution environment. An IDA may also specify **ports** each of which defines a connection to be established from the IDA to a **window** on a neighbouring **component**; such **ports** allow data to be projected from one IDA to another with no intervening **activity**. A **composite** form of IDA is provided for network decomposition in terms of internal IDAs.

Implementation part The part of a **template module** which defines the internal details of the **template**. For **simple templates** it defines the data and algorithms together with references to the required **definitions** and **library interfaces**. For **composite templates** it defines the **components** and their interconnections. For **simple templates** this corresponds to the information necessary to achieve **fully enrolled status**. For **composite templates** it corresponds to the information required to achieve at least **partially enrolled status**.

Interface A **specification** for a connection between two **components**. There are three types of interface: **access interface**, **library interface**, **subroot interface**. An interface may refer to one or more **definitions** to import supplementary **specifications**.

Introduce The introduce operation checks that the **name part** and **specification part** of a **module** have been defined and are legal. If the checking is successful, then the **module** will be accorded **partially** or **fully introduced status**, depending upon the **status** of the **modules** to which it refers in its **specification part**.

Library The Mascot design entity implementing the set of procedures specified in one or more **library interfaces**. This **module** supports procedural decomposition of **simple templates** and must be capable of multi threaded operation. It differs from an **IDA** which is also multi threaded in that there is no interaction between the threads.

Library Interface A **simple specification** defining the operations which a **library** makes available to any **simple template**.

Link A control flow connection from one **subelement** to another. Its diagrammatic representation is a line which carries a hollow arrowhead to indicate the direction of invocation. The type of a link is a **subroot interface** which defines the nature of the interactions between these **subelements**.

Mascot Database The collection of all Mascot **modules**, their **status** and their derived products known to a particular support environment.

Module A textual unit (or possibly a set of such units) representing a **specification** or a **template**. It possesses an explicit **class** which reflects its contents. In their completed form, all **classes** of module have a **name part** and a **specification part**, and in addition the **template** modules have an **implementation part**.

Name part The section of a **module** which defines the **class** and name of the **template** or **specification** which the **module** represents.

Network A set of interconnected Mascot **components** (**elements** and other networks) which constitutes the whole (**system**) or part (**subsystem**, **composite IDA**, **composite server**) of a Mascot application.

Partially Enrolled A **status** value given to a **composite template module** for which the **enrol** operation was successful, but which failed to achieve **fully enrolled status**. It indicates that at least one of the **templates** which define the **module**'s **components** has not yet achieved **fully enrolled status**.

Partially Introduced A **status** value given to a **module** for which the **introduce** operation was successful, but which failed to achieve **fully introduced status**. It indicates that at least one of the **specification modules** referred to in its **specification part** has

not yet achieved **fully introduced status**.

Path A data flow connection between a **port** of one **component** and a **window** of another. Its diagrammatic representation is a line which normally carries a solid arrowhead to indicate the direction of data flow. The type of a path is an **access interface** which defines the nature of the interactions between these **components**. A **composite** form of path is available, where the type of the path is a **composite access interface**.

Pool A special case of an **IDA** having destructive write properties. Data in a pool is overwritten on a write operation but may be read repeatedly.

Port A named reference to an **access interface** by means of which a **template** expresses its requirement for interactions with other **templates**. Externally it expresses a connectivity constraint of the **template**. Internally, the port name is used as a means of selecting between ports. A port is the active end of a **path**. Its diagrammatic representation is a small filled circle on the boundary of a **component**. A **composite** form of port is available.

Registered A **status** value indicating that the **name part** of a **module** has been defined and is legal.

Root The Mascot design entity representing the **subelement** which contains the initial entry point of a **composite activity**. A root usually specifies one or more **subroot interfaces** which define connections to be established to **subelement components** in the same **activity**. A root may also specify one or more **ports** each of which defines a connection to be established from the root **component** to a **port** of the enclosing **activity**. A root **module** is a **template** which may be used to create root **components**.

Server The Mascot design entity representing a single independent device handling **element**; it is the only form of design entity which can be used for this purpose. It has all the features of an **IDA** and in addition may contain one or more **handler** routines to be invoked as a direct consequence of hardware interrupts. A server **module** is a **template** which may be used to create server **components** each of which allows several independently scheduled single sequential program threads to be simultaneously active or suspended within the server. A **composite** form of server is provided for network decomposition in terms of internal **servers** and **IDAs**.

Simple A simple **module** is one which is not further decomposed in terms of lower level **modules**. Some **templates** and all **specifications** have a simple form which is described in detail under the appropriate heading.The following **templates** have simple forms: **activity, IDA, server, root, subroot, library**.

Specification A **module** defining an **interface** or **definition**. All have a **simple** form but only an **access interface** may be **composite**.

Specification part The part of a **module** which specifies the external view of the **template** or **specification** which it represents. In the case of a **specification**, it completes the **module**. In the case of a **template**, it contains sufficient information for **components** of that type to be included in a **composite template**. It also constrains the **implementation part**. It corresponds to the information required to achieve at least **partially introduced status**.

Status An attribute associated with a **module** indicating the degree of progress made in the definition and checking of the **module** and of any other **modules** to which it refers, and hence indicating its fitness for use by other **modules**. Five status values are defined: **registered, partially introduced, fully introduced, partially enrolled** and **fully enrolled**.

Subelement A Mascot entity supporting sequential decomposition of an **element** or **subelement** (ie a **root** or **subroot**).

Subroot The Mascot design entity representing a **subelement**. A subroot specifies a single **subroot interface** defining the connections which may be established from other **subelement components** in the same **activity**. A subroot may specify one or more **subroot interfaces** which define connections to be established to other **subelement components** in the same **activity**. A subroot may also specify one or more **ports** each of which defines a connection to be established from the subroot **component** to a **port** of the enclosing **activity**. A subroot **module** is a **template** which may be used to create **subelement components**. A **composite** form of subroot is provided for decomposition in terms of internal **subroots**.

Subroot Interface A **simple specification** defining the possible interactions (eg procedure specifications) between the **components** connected by a **link**.

Subsystem A **network** representing part of a Mascot application. Subsystems may possess both **ports** and **windows** and may therefore communicate directly with each other. Where a **subsystem** contains no **activity**, either directly or indirectly, it is functionally identical to a **composite IDA** or a **composite server**.

System A **network** representing the outermost level of software description. When supported by all the **modules** referred to explicitly or implicitly, it constitutes a complete Mascot application software description.

Template A pattern from which **components** may be created during **building**. The creation of these **components** is controlled by definitions within **composite** templates.

Template Constant A constant, named in the **specification part** of a **template**, for use within the **implementation part**. Its value is usually supplied when **components** to be created from the **template** are named in a (higher level) **composite template**. In the case of a **system template**, the value is supplied as part of the **building** process.

Window A named reference to an **access interface** by means of which a **template** expresses the interactions it provides for use by other **templates**. Externally, it expresses a connectivity constraint of the **template**. Internally, the window name is used as a means of allocating internal features between windows. A window is the passive end of a **path**. Its diagrammatic representation is a small filled rectangle on the boundary of a **component**. A **composite** form of window is available.

UNLIMITED