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

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**.

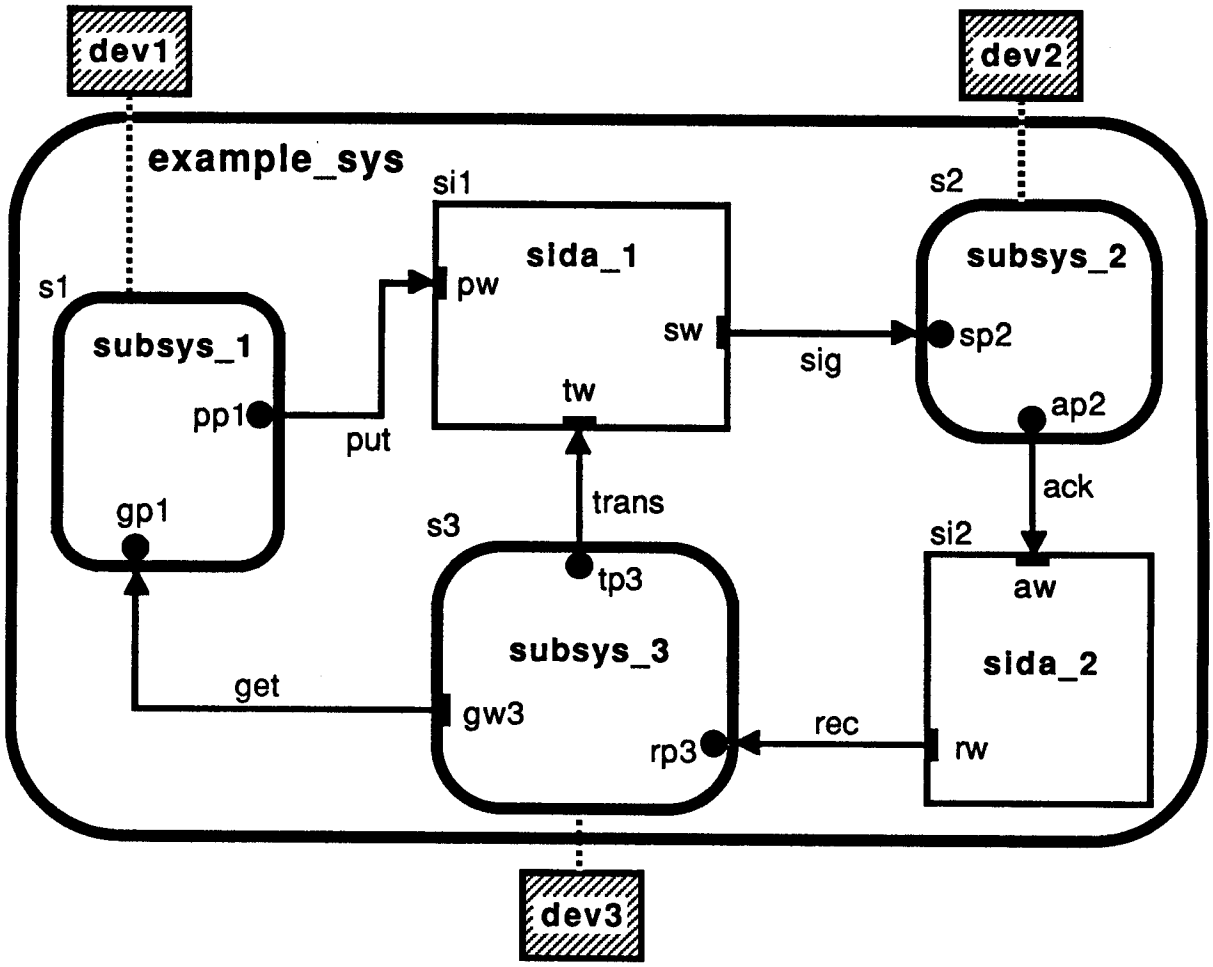
The hierarchical nature of this structure permits the design to be viewed at various levels of abstraction, examination of any one of which immediately highlights the second salient feature of the Mascot design representation. This is data flow. Each level of the design is conceived as a network through 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.



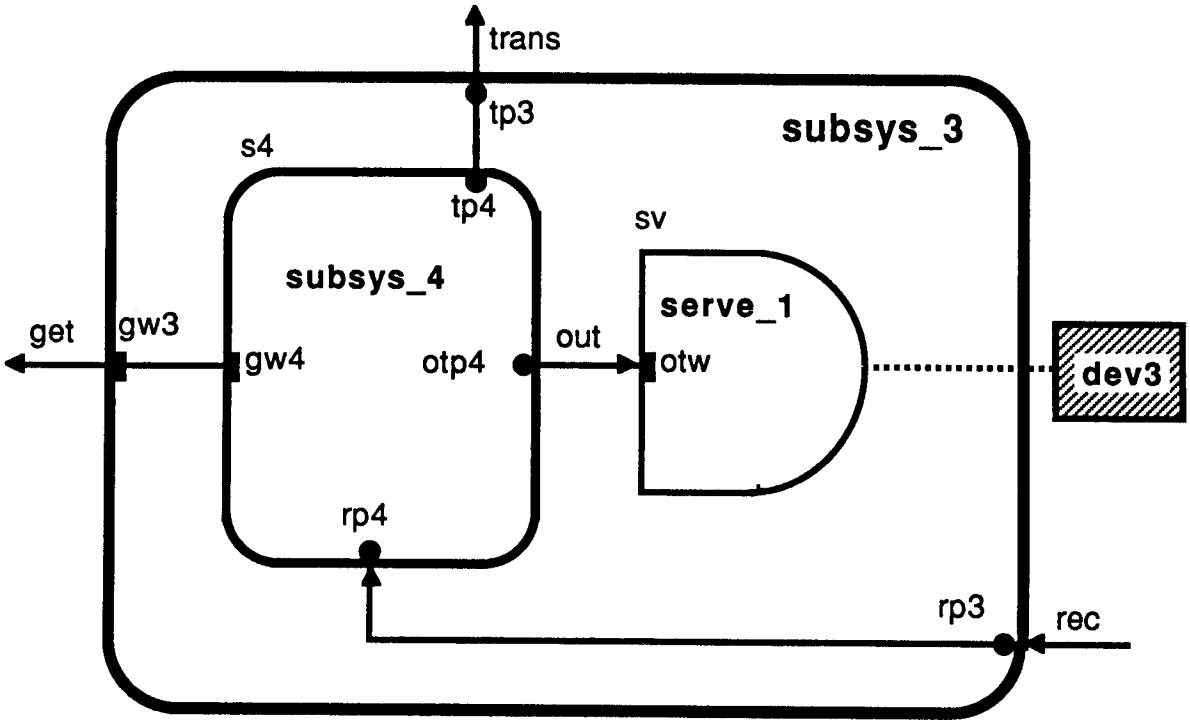
This diagram shows our example **system**. It corresponds to what, in the Mascot method, is called the 'initial design response'. It shows, on the outside of the **system**, the set of hardware devices the **system** is required to interact with. It shows, on the inside, a set of high level design components 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

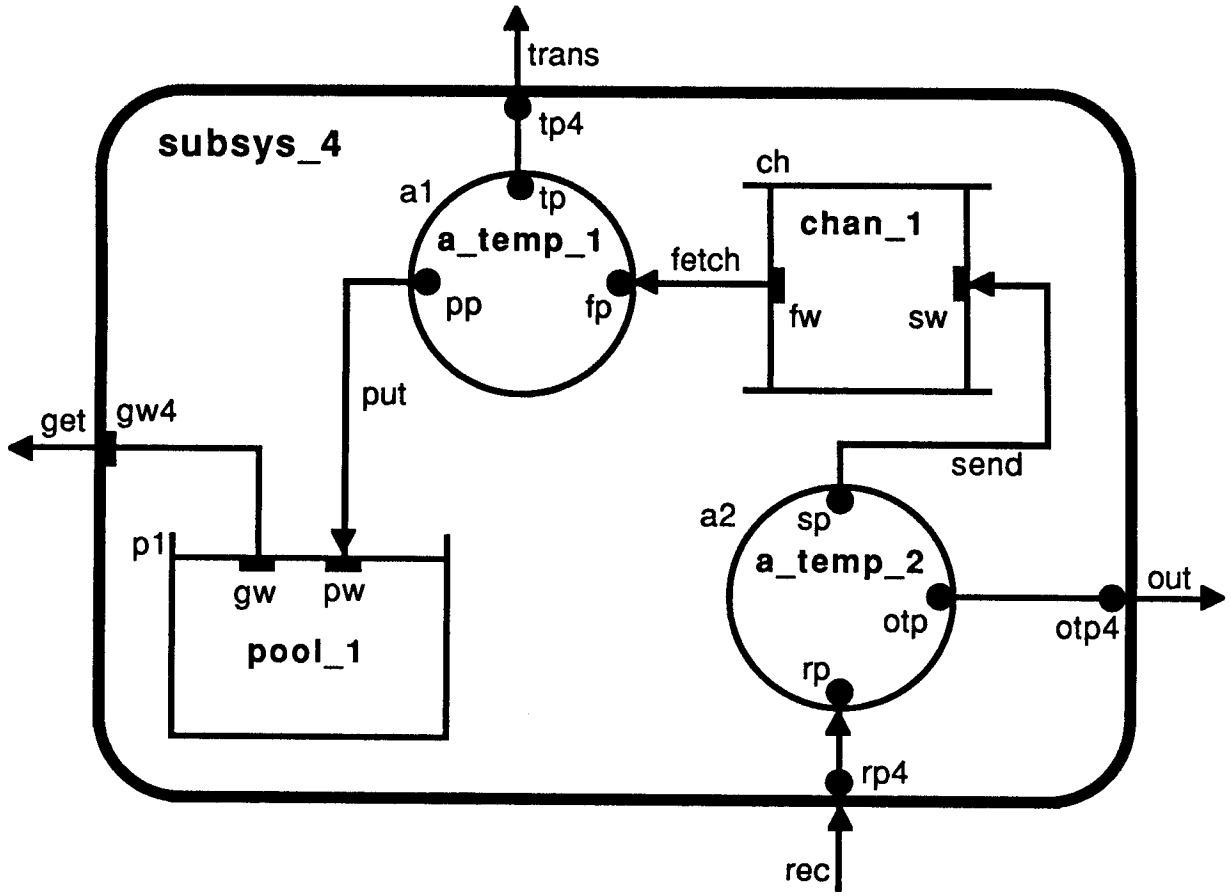
fundamental mechanisms are not made directly available by the run-time system. Instead, a set of higher level operations/facilities such as monitors, message passing or rendezvous are provided. In Mascot the view was taken that, in order to obtain the optimum performance which is always vital in in embedded 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 **IDAs** contain data which can be the subject of interaction from several **activities** and that only **IDAs** 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 **IDAs** 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 **IDAs**. 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.

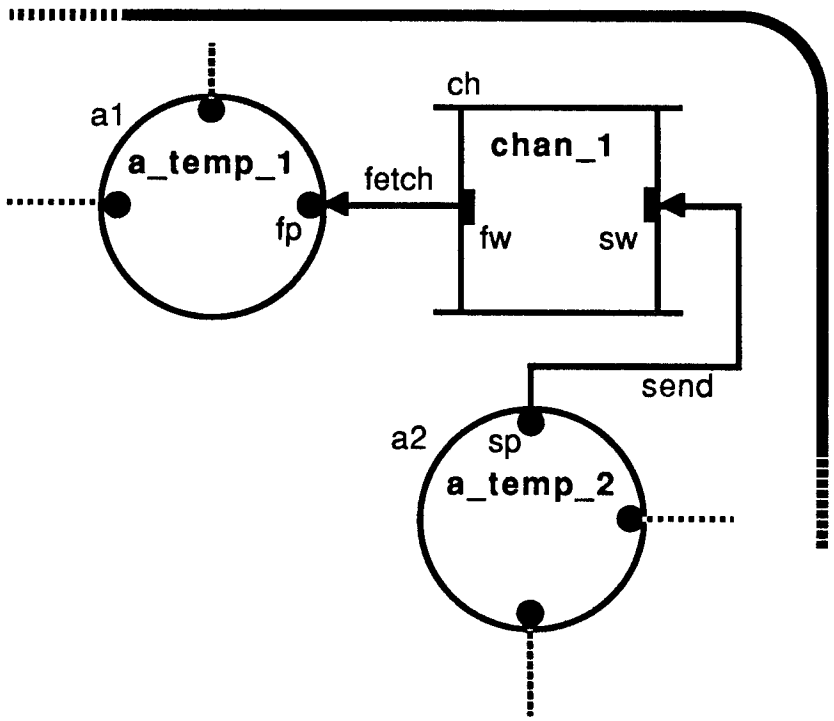
The **template**, ***subsys_4*** also contains two **IDAs**. They are represented by slightly modified versions of the simple rectangular symbol seen earlier in the **system** diagram. This shows them to be special cases corresponding to the **channels** and **pools** of previous versions of Mascot. 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 ) ;
```

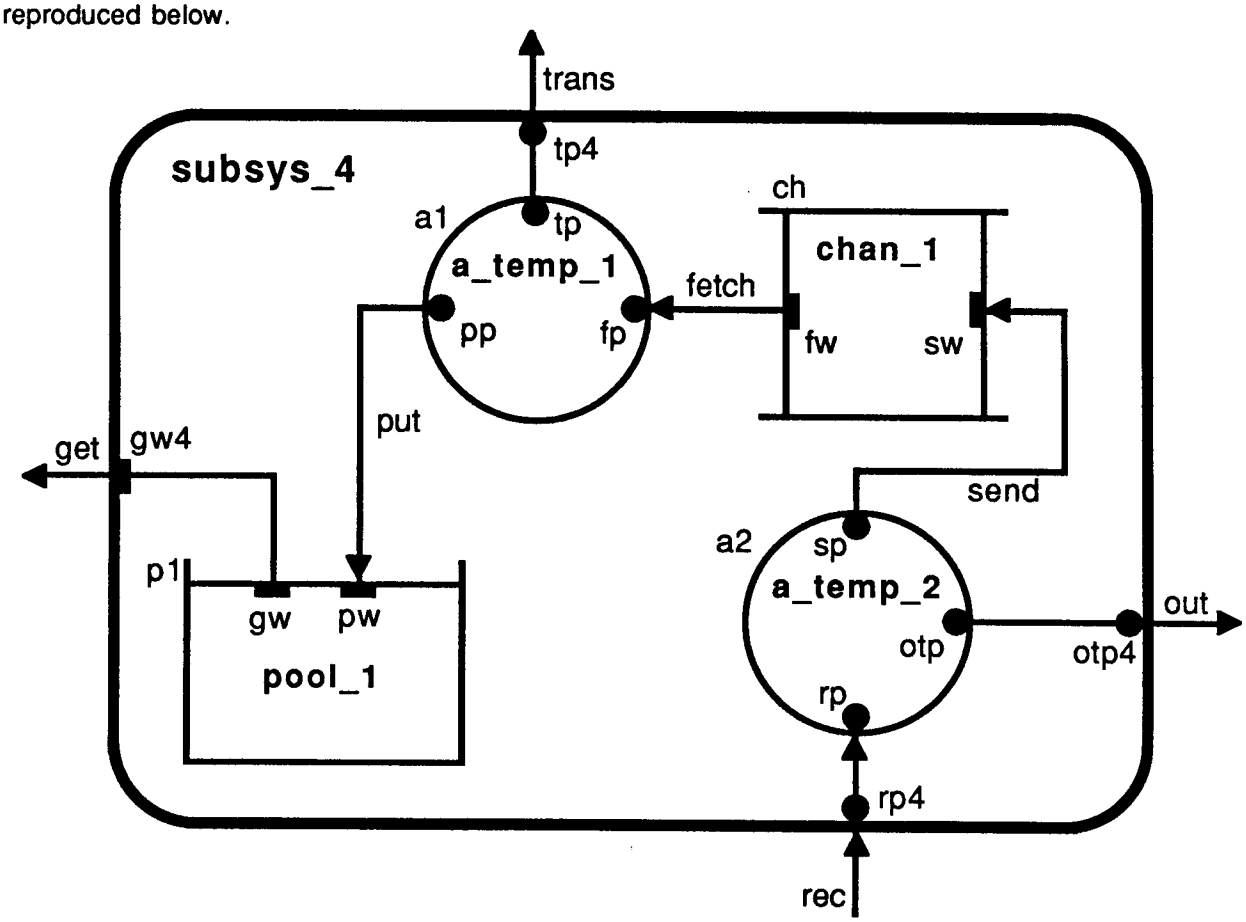
With the interconnections as described, **activity component *a2*** would, by this means, invoke execution of procedure ***insert*** in the **IDA component *ch.*** Similarly **activity *a1*** would invoke procedure ***remove*** (recall the access equivalence list) of ***ch*** by:

```
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 **IDAs** 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***

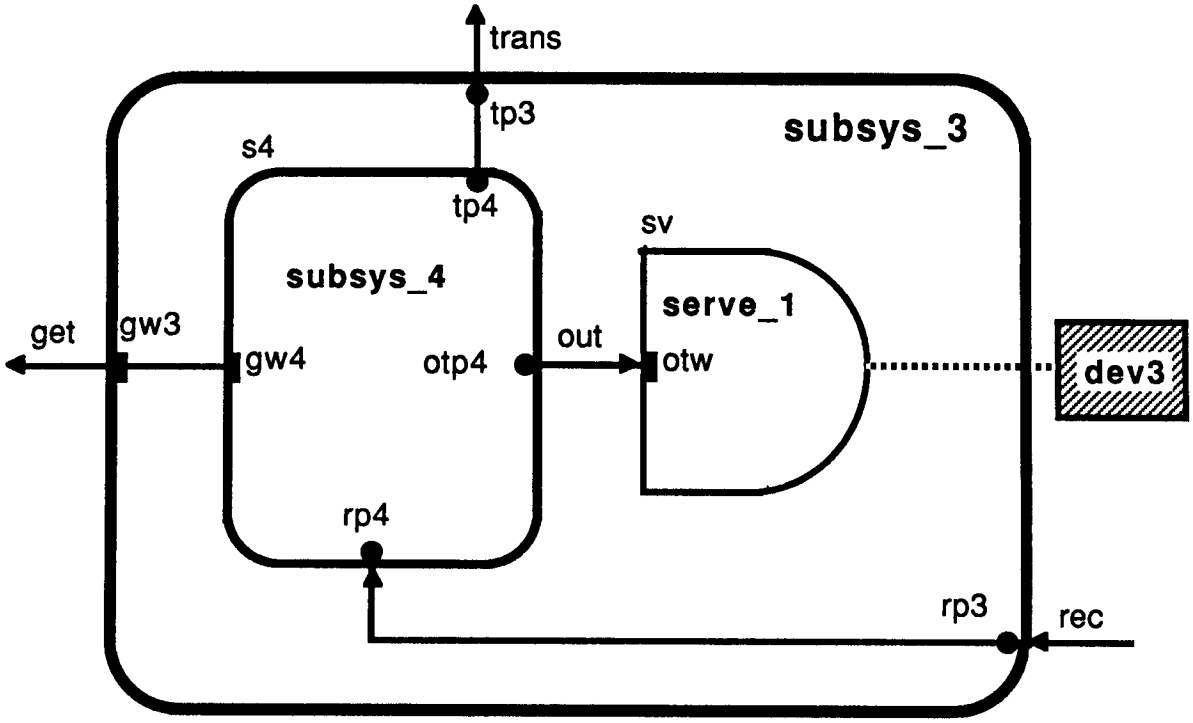
port ***otp*** of ***a2*** <----------> **port** ***otp4*** of **template**

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***.