INTERNATIONAL STANDARD

ISO 9074

First edition
1989-07-15

AMENDMENT 1
1993-11-15

Information processing systems — Open Systems Interconnection — Estelle: A formal description technique based on an extended state transition model

AMENDMENT 1: Tutorial on Estelle

Systèmes de traitement de l'information — Interconnexion de systèmes ouverts — Estelle: Technique de description formelle basée sur un modèle de transition d'état étendu

AMENDEMENT 1: Tutorial sur Estelle

Reference number
ISO 9074:1989/Amd.1:1993(E)

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Amendment 1 to International Standard ISO 9074:1989 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Sub-Committee SC 21, *Information retrieval, transfer and management for open systems interconnection (OSI)*.

Information processing systems - Open Systems Interconnection - Estelle: A formal description technique based on an extended state transition model

AMENDMENT 1: Tutorial on Estelle

Insert a new annex D as follows:

ANNEX D

(informative)

Estelle Tutorial

INTRODUCTION

Estelle is a formal description technique (FDT) for specifying distributed, concurrent information processing systems with a particular application in mind, namely that of communication protocols and services of the layers of Open Systems Interconnection (OSI) architecture defined by ISO.

Estelle is a second generation formal description technique and it reflects the experience gained from experiments in using its predecessors (see [4], [5], [8], [9] and [56]). Estelle also reflects collaboration with ITU-T, which defined SDL (Specification and Descriptions Language [13]) with which Estelle has some notions in common.

Estelle is one of the description techniques that are intended to serve as means to remove ambiguities from ISO protocol standards, traditionally defined by a combination of a natural language prose, state tables, etc. However, an unambiguous formal specification still may be far from any implementation. There is a vital need for formalised specifications of distributed systems in general, and communication protocols in particular, which would, at the same time, indicate how implementations may be derived from them. This is precisely the principal field of application of Estelle.

Estelle can be briefly described as a technique that is based on an extended state transition model, i.e., a model of a non deterministic communicating automaton extended by the addition of Pascal language. More precisely, Estelle may be viewed as a set of extensions to ISO Pascal [41], level 0, which models a specified system as a hierarchical structure of communicating automata which:

- may run in parallel, and
- may communicate by exchanging messages and by sharing, in a restricted way, some variables.

Estelle permits a clear separation of the description of the communication interface between components of a specified system from the description of the internal behaviour of each such component. As in Pascal, all

manipulated objects are strongly typed. This property enables static detection (e.g. during compilation) of specification inconsistencies.

D.1 A brief overview of the principal concepts in Estelle

D.1.1 Modules and module instances

An Estelle specification describes a collection of communicating components. Each component is in fact an instance of a module defined within the Estelle specification by a module definition. Thus, it is appropriate to call components *module instances*. Hereafter, the term *module* will be used rather than module instance unless it can lead to confusion.

The behaviour of a module and its internal structure are specified respectively by the set of transitions (of an extended state transition model) that the module may perform and by the definition of submodules (children modules - see D.1.2) of the module together with their interconnections (see D.1.3).

A module is *active* if its definition includes, in its transition part, at least one transition; otherwise, it is *inactive*.

In Estelle particular care is taken to specify the communication interface of a module. Such an interface is defined using three concepts:

- *interaction points;*
- *channels;*
- *interactions*.

A module may have a number of input/output access points called *interaction points*. There are two categories of interaction points: *external* and *internal*. With each interaction point a *channel* is associated that defines two sets of *interactions*. These two sets consist of interactions that can transmitted and received, respectively, through an interaction point. Interactions are abstract events (messages) exchanged with the module environment (through external interaction points) and with children modules (through internal interaction points).

Informally, a module will be represented graphically as a box (rectangle) possibly with points on its boundary (external interaction points) and inside of it (internal interaction points). The module name and its class attribute (see D.1.4), the names of interaction points and their associated interactions (going in or out) may be added as shown in figure 1.

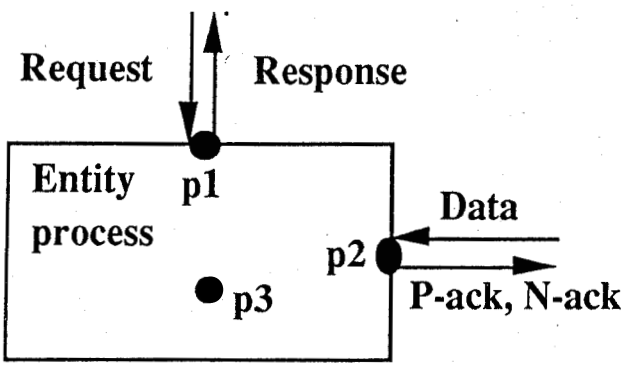

Figure A.1 - Graphical representation of a module

D.1.2 Structuring of modules

A module definition in Estelle may textually include definitions of other modules. Applied repeatedly, this leads to a hierarchical tree structure of module definitions.

Informally, the hierarchical tree structure of modules may be depicted as in figure A.2a or as in figure A.2b. The modules are represented by boxes. The parent/child relationship is represented either by an edge or by nesting of boxes. The root of the tree (or the largest enclosing box) is the specification (main) module representing the whole specification. It is assumed that one (and only one) instance of the specification module always exists.

The hierarchical tree structure of modules constitutes a pattern for any hierarchy of module instances. The hierarchical position of a module instance corresponds to the position of the module definition in this pattern. By definition the specification module corresponds to one and only one module instance. Any other module may correspond to any number of instances. This number may change dynamically (see D.1.5).

The hierarchical tree structure of module instances that may correspond to the hierarchical tree structure of modules is depicted in figure A.2c or as in figure A.2d.

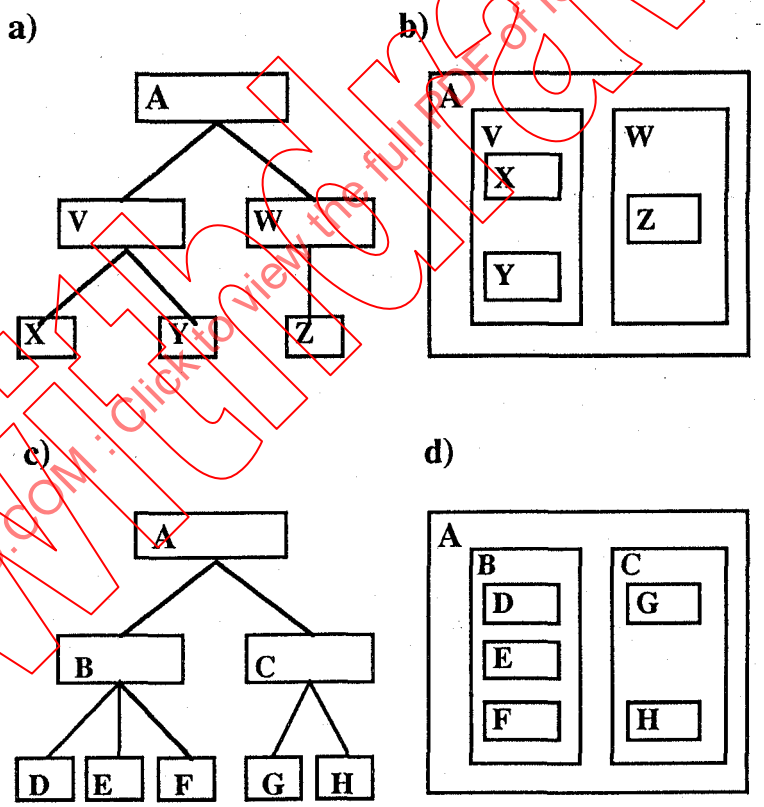

Figure A.2 - Graphical representation of a hierarchy of modules ((a) and (b)) and of a hierarchy of their instances ((c) and (d)).

Children of the same parent are called *siblings* (e.g., modules V and W in figure A.2). The transitive relationship between modules in a hierarchy are called *ancestors* and *descendants* (e.g., module A is the ancestor of module X and module X is the descendant of module A in figure A.2).

D.1.3 Communication

Module instances within the hierarchy can communicate by

- message exchange;
- restricted sharing of variables.

D.1.3.1 Message exchange

The module instances may exchange messages, called *interactions*. A module instance can send an interaction to another module instance through a previously established communication link between their two interaction points. An interaction received by a module instance at its interaction point is appended to an *unbounded* FIFO queue associated with this interaction point. The FIFO queue either belongs exclusively to the single interaction point (**individual queue**) or is shared with some other interaction points of a module (**common queue**).

A module instance can always send an interaction. This principle is sometimes known as *non-blocking send* (or *asynchronous* communication) as opposed to *blocking-send* also known as *rendez-vous* (or *synchronous* communication).

To specify which modules are able to exchange interactions, *communication links* between modules' interaction points are specified by means of **connect** and **attach** operations.

A communication link between two interaction points is composed of exactly one *connect segment* and zero or more *attach segments*. Informally, each link segment (connect or attach) will be represented graphically by line segments which bind modules' interaction points. Figure A.3 illustrates this convention.

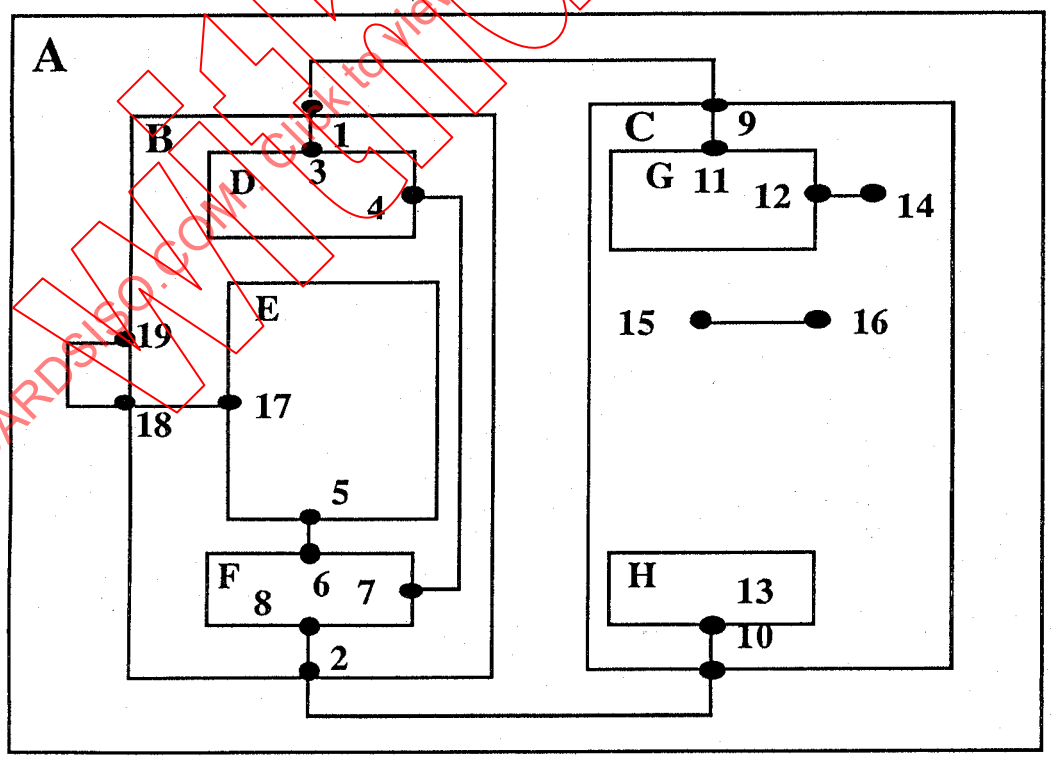

Figure A.3 - Module instances and their communication links.

When an external interaction point of a module is bound to an external interaction point of its parent module, we say that these interaction points are *attached*. In figure A.3 the following pairs of interaction points are attached: (1, 3), (2, 8), (9, 11), (10, 13) and (17, 18).

Two bound interaction points are said to be *connected* if both are external interaction points of two sibling modules (e.g., (1, 9), (2, 10), (4, 7), and (5, 6) in figure A.3), or one is an internal interaction point of a module and the other is an external interaction point of one of its children modules (e.g., (12, 14) in figure A.3), or both are internal or external interaction points of the same module (e.g., (15, 16) and (18, 19) in figure A.3).

The specific restrictions which are imposed on connections and attachments of interaction points are detailed in D.3.

Note also that an interaction point definition does not determine how the interaction point must be bound. Two instances of the same module may have the corresponding interaction point bound differently. For example, module instances E1 and E2 in figure 5 may be instances of the same module but their corresponding interaction points are bound differently.

The communication link specifies that two module instances whose interaction points are the *end-points* of the link can communicate by exchanging messages (in both directions) through these linked interaction end-points. In figure A.3, for example, interaction points 3 and 11 or 8 and 13 are end-points of links between modules D and G, and F and H, respectively. The interaction points 1, 9, 2 and 10 are not end-points. Note that, at a given moment, an interaction end-point may be linked to at most one other interaction end-point.

If a module outputs (sends) an interaction through an interaction point that is an end-point of a communication link then this interaction directly arrives at the other end-point of this link and is stored in an unbounded FIFO queue of the receiving module. If a module outputs an interaction through an interaction point that is not an end-point of a communication link then the interaction is considered to be discarded. Thus only end-to-end communication between modules' interaction points is possible.

Several communication links may, however, simultaneously exist between the interaction points of a given module instance and interaction points of other module instances. Thus *multicast* communication may be specified. For example the module A in figure A.4 may multicast an interaction by sending it simultaneously (in one transition) through its interaction points p[1], p[2] and p[3] to modules A1, A2 and A3. Observe that in Estelle all three of these interaction points may be declared as elements of an array (see D.2.1).

D.1.3.2 Restricted sharing of variables

Certain variables can be shared between a module and its parent module. These variables must be declared as *exported variables* by the module. This is the only way variables may be shared. The simultaneous access to these variables by both the module and its parent is excluded because the execution of the parent's actions always has priority (the *parent/children priority principle* of Estelle - see also D.4.3.2).

Note that sharing variables is not the only way of communication between a parent and its child: they may also communicate by message exchange (see for example communication links between interaction points (12, 14) and (17, 19) in figure A.3).

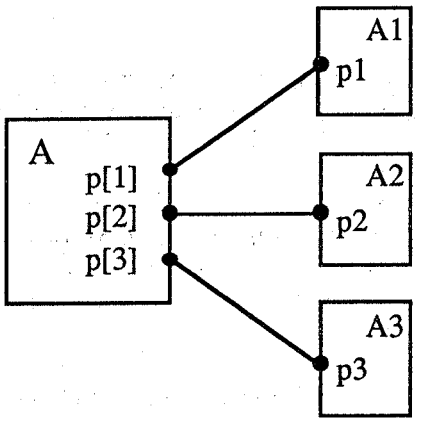

Figure A.4 - Multicast communication

D.1.4 Parallelism and Nondeterminism

The way module instances behave with respect to each other is strictly dependent on the way they are nested (see D.1.2) and attributed.

A module may have one of the following *class* attributes

- systemprocess,
- systemactivity,
- process,
- activity,

or may be not attributed at all.

All instances of a module have the same attribute as that defined for the module in the module's header definition.

The modules attributed with **systemprocess** or **systemactivity** are called *system* modules.

The following five *attributing principles* must be observed within a hierarchy of modules

(a) Every active module (i.e., such that its definition includes at least one transition) shall be attributed,

(b) System modules shall not be nested within an attributed module,

(c) Modules attributed with **process** or **activity** shall be descendants of a system module,

(d) Modules attributed with **process** or **systemprocess** may be substructured only into modules attributed with either **process** or **activity**,

(e) Modules attributed with **activity** or **systemactivity** may be substructured only into modules attributed with **activity**.

Observe that inactive modules may be attributed. Observe also that all modules embodying a system module are inactive and nonattributed, and that those are the only nonattributed modules within the hierarchy.

The attributes **systemprocess** and **systemactivity** serve to identify separate communicating systems within the specification. In particular the specification itself may have one of these attributes. In such a case the specification describes one system. Each system is a subtree of modules rooted at the system module. The number of system instances within a specification is always fixed.

For clarity of presentation, the following conventions are assumed in subsequent figures:

- system modules (system roots) and their communication links are in bold lines (these form a ***static*** system architecture),
- dotted lines are used for modules enclosing systems modules,
- non bold lines are reserved for remaining modules and links.

Figure A.5 illustrates these conventions. Module A is an unattributed specification (main) module. It has two children (system modules) B and C attributed with **systemprocess** and **systemactivity**, respectively. Module B has three children: D, E and F, attributed with **process**, **activity** and **process**, respectively. Module C has two children G and H both attributed with **activity**. Module E has two children both attributed **activity**. Within the above specification two systems are identified. Each system is a subtree of module instances rooted at a system module (modules B and C).

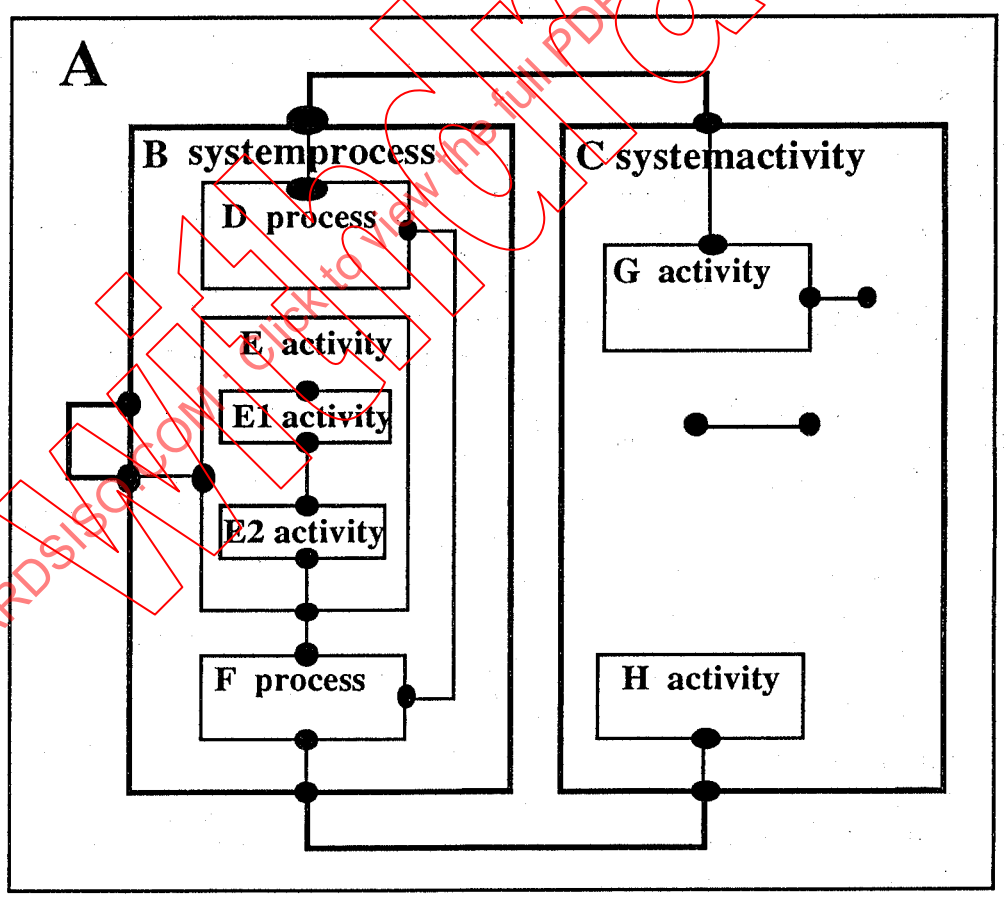

Figure A.5 - Attributed module instances and their communication links; different lines are used to represent system modules (bold), modules enclosing system modules (dotted) and others (non bold)

The attributes of modules play an important role in defining the behaviour of a specification (see also the examples in D.4.4).

An attributed module instance acts as a supervising-like manager of its children instances. Recall (see D.1.2) that all ancestor (enclosing) modules of a system module are nonattributed. This means that system modules do not have any supervisor and thus all means of control of their respective behaviour is absent. The systems run in a *parallel asynchronous* way with respect to each other.

Within a system, one of two possible behaviours among the system's module instances may be specified by means of the attribute assigned to the system module:

- a *synchronous parallel* execution, when the **systemprocess** attribute is assigned,
- a *nondeterministic* execution, when the **systemactivity** attribute is assigned.

D.1.5 Dynamism within a system

The system instances and their interconnections (connections of their interaction points) once created (by executing specification *initialisation part*) are fixed forever (are invariant). This is due to the fact that modules enclosing system modules are always inactive (do not have any transitions) and thus do not have any means to dynamically change the system configuration. It is possible, however, that different invariant (static) structures may be created due to the fact that within the specification different ways of initialising it may be defined (see D.2.4).

In contrast the internal structure of each system and the bindings between interaction points of their submodules may vary (i.e., both are *dynamic*). This is because actions (transitions) of an active module instance within a system may include statements creating and destroying its children and also creating and destroying bindings (attachments or connections) between interaction points of children or between the interaction points of the module instance and its children. For example, an action of the module instance E (figure A.5) may create or destroy its children module instances E1 and E2 as well as the connection between E1 and E2 and the attachment between E and E2 (compare figure A.5 with figure A.3).

Recall that although the number of instances of a specific module may change in the dynamic structure of an Estelle specification, the hierarchical position of each instance corresponds to the respective position of its module definition in the specification.

D.1.6 Typing

All manipulated objects are strongly typed. Pascal typing mechanisms are extended to purely Estelle objects such as: module variables, interactions, interaction points and (control) states.

D.1.7 Module internal behaviour

The internal dynamic behaviour of an Estelle module is characterised in terms of a nondeterministic extended state transition model, i.e., by defining a set of *states*, a subset of *initial states* and a *next-state relation*.

An extended state is, in general, a complex structure composed of many components such as: value of the control state, values of variables, contents of FIFO queues associated with interaction points and the status

of the module internal structure (submodule instances, bindings between interaction points, etc.). Initial states of a module instance are defined by an initialisation part of the module definition.

The next-state relation of a module instance is defined by a set of transitions declared within a transition part of the module definition. Each transition definition contains necessary conditions enabling the transition execution, and an action to be performed when it is executed. An action may change the module instance's state described above and may output interactions to the module environment. A compound-statements of Pascal is used to define the actions of a transitions .

The execution of a transition by a module instance is considered to be an *atomic* operation. This means that once a transition's execution is started, it cannot be interrupted, and conceptually, one cannot observe intermediate results.

The well-known model of a finite state automaton (FSA) is a particular case of a state transition system. Hence, an FSA may be described in Estelle (see D.4.2).

D.1.8 Global behaviour

To describe the global behaviour of an Estelle specification, the operational style (operational semantics) has been used.

The global behaviour is defined by the set of all possible sequences of *global situations* generated from an *initial situation*. Two consecutive global situations correspond to the execution of one transition (recall that transitions are atomic)

The operational semantics for Estelle describes the way these sequences are generated, i.e., the way the system's transitions (transitions of its modules) may be interleaved to properly model synchronous parallelism within subsystems combined with asynchronous parallelism between them. This global semantics model is described in more detail in D.4.3.

The notion of time appears in Estelle to interpret properly "delays" (i.e., dynamic values assigned to some transitions which indicate a number of time units by which the execution of these transitions must be delayed). However, the semantic model retains the hypothesis that execution times of transitions are unknown. This knowledge is considered implementation dependent. The model of Estelle outlined above is dependent on a time process, which is assumed to exist, only in that a relationship between progress of time and computation is defined and the *delay-timers* are observed to decide whether a transition can or cannot be fired. The way the delay-timers are interpreted is explained by an example in D.4.2.

The constraints formulated specify a class of acceptable time processes. In each implementation or for simulation purposes, any element of this class may be chosen. (See [30] and [25]).

D.2 Syntax and interpretation of Estelle concepts

D.2.1 Channels, interactions and interaction points

Channels in Estelle specify sets of interactions (messages). Declarations of interaction points refer to channels in a specific way. By such a reference, a particular interaction point has a precisely defined set of interactions that can be respectively sent and received through this point (in a way, the interaction points are typed). Consider, for example, the following channel definition

```
channel CHANNEL_Id(ROLE_Id1, ROLE_Id2);
   by ROLE_Id1:    m1;
                   m2;
                   :
                   mN;

   by ROLE_Id2:    n1;
                   n2;
                   :
                   nK;

   by ROLE_Id1, ROLE_Id2:   k1;
                            k2;
                            :
                            kP;
```

where m1,..,mN, n1,..,nK, k1,k2,..kP are interaction declarations.

Each interaction declaration consists of a name (interaction-identifier) and possibly some typed parameters. Thus, an interaction declaration

REQUEST(x: **integer**; y: **boolean**)

specifies in fact a class of interactions (an interaction type) with a common name REQUEST. Each of the interactions in the class is obtained by a substitution of actual parameters (values) for formal parameters x and y. Therefore,

REQUEST(1, true) and REQUEST(3, false)

are both interactions in the class specified by the interaction declaration above. In the absence of parameters, the interaction-identifier represents itself.

Now, an interaction point p1 may be declared as follows

p1 : CHANNEL_Id(ROLE_Id1)

and another interaction point p2,

p2 : CHANNEL_Id(ROLE_Id2)

In the first case, the set of interactions that can be sent via p1 contains all interactions specified after "by ROLE_Id1" and after "by ROLE_Id1, ROLE_Id2" in the channel definition (i.e., the interactions declared by m1,m2,..,mN and k1,k2,..kP), and the set of interactions which can be received contains all interactions specified after "by ROLE_Id2" and after "by ROLE_Id1, ROLE_Id2" in the channel definition (i.e., the interactions declared by n1,n2,..,nK and k1,k2,..kP). Observe that interactions declared for both roles (i.e., after "by ROLE_Id1, ROLE_Id2") are those that can be sent and received (and they have to be declared in this way).

In the second case we have, as it is easy to guess, an exact opposite assignment of interactions sent and received, i.e., those interactions which could be previously sent via p1 can now be received via p2 and vice versa.

We say that interaction points p1 and p2 above play *opposite roles* (or have *opposite types*). Two interaction points both referring to the same channel and the same role-identifier are said to play the *same role* (or have the *same type*).

Two interaction points that are linked (in particular, connected) shall play opposite roles since the exchange of interactions takes place between them (any interaction sent via one interaction point is received via the second and vice versa). Two interaction points that are attached must play the same role since the aim of attaching them is to "replace" one of them by the other.

Finally, to specify whether the interaction point p1 does or does not share its FIFO queue with other interaction points we respectively write:

p1 : CHANNEL_Id(ROLE_Id1) **common queue**

or

p1 : CHANNEL_Id(ROLE_Id1) **individual queue**

In the former case, the FIFO queue will be shared with all those interaction points (external or internal) which were declared **common queue** within the module.

A group of interaction points of the same type may have a common declaration by means of an array construct. For example,

p : **array**[1..3] **of** CHANNEL_Id(ROLE_Id1)

specifies, in fact, three interaction points referenced by p[1], p[2] and p[3].

Both external and internal interaction points of a module are declared in the way described above. The distinction is made only by virtue of where they are declared. External interaction points of a module are declared within its module header definition (see D.2.2), while internal interaction points are declared within the declaration part of its body definition (see D.2.3).

D.2.2 Modules

A **module** definition in Estelle is composed of two parts:

- a module header definition, and
- a module body definition.

A *module-header* definition specifies a *module type* which identifies a class of modules with the same external visibility, i.e., with the same interaction points and exported variables, and the same class attribute.

The definition of a module header begins with the keyword **module** followed by its name and optionally by: a class attribute (**systemprocess**, **process**, **systemactivity** or **activity**), a list of formal parameters, and declarations of external interaction points (after the keyword **ip**) and exported variables (after the keyword **export**). The definition finishes with the keyword **end**. The actual values of the formal parameters are assigned when a module instance of the module header type is created (initialised) - see D.2.4.

The following is an example of a module header definition:

```
module A systemactivity (R: boolean);
    ip   p    : T(S) individual queue;
         p2   : W(K) common queue;
         p3   : U(S) common queue;
    export  X, Y : integer;  Z : boolean
end;
```

Observe that, by the above definition, the same FIFO queue is associated with (is shared by) the interaction points p1 and p2, which means that any interaction received through p1 or p2 will be appended to the (common) queue.

Usually one *module body* definition is declared for each module header definition. However, more than one body may be declared for a header definition to specify possibly different internal behaviour and substructure.

A module body definition begins with the keyword **body** followed by: the body name, a reference to the module header name with which the body is associated, and either a body definition followed by the keyword **end** or the keyword **external**.

For example, the following two bodies may be associated with the module header A:

```
body B for A;
{body definition  see D.2.3}
end;

and

body C for A; external;
```

In fact, at a conceptual level, two modules have been defined: one of which may be identified by the pair (A, B), and the second by the pair (A, C). The modules thus defined have the same external visibility (same interaction points p, p1, p2 and same exported variables X, Y, Z) and the same class attribute (systemactivity). But their behaviours, defined by the body definitions are, in principle, different. This means that modules may have different behaviours and the same external visibility. A body defined as **external** does not denote any specific behaviour of the module. It indicates that either the module body definition already exists elsewhere or will be provided later while refining the specification. The "external" bodies nicely serve to allow describing an overall system architecture without any detailed description of the system components. This feature is illustrated by the example in 4.1.

The body definition is composed of three parts:

- a declaration part;
- an initialisation part;
- a transition part.

D.2.3. Declaration part

The declaration part of a body definition contains usual Pascal declarations (constants, types, variables, procedures and functions) and declarations of specific Estelle objects, namely:

- channels;
- modules headers and bodies;
- module variables;
- states and state-sets;
- internal interaction points.

Note that, unlike in Pascal, all these declarations may appear in any order and even several times. Note also, that *undefined* types (e.g., **type** buffer = ...) and constants defined using **any** construct (e.g., **const** T = **any** INTEGER) may be declared. These two facilities are introduced to allow refinements of the specification.

A body definition which is being declared may contain declarations of other modules (headers and bodies). This, applied repeatedly, leads to a hierarchical tree structure of module definitions.

For example, the body definition B declared below for a module-header A contains definitions of modules (A1, B1) and (A1, B2). These are children modules of the module (A, B), where the detailed definition of the module header A is that from the previous clause D.2.2. The hierarchy of the module definitions is depicted in figure 6.

```
module A (* see D.2.2 *)..end;
body B for A;
   ip p1 : T1(R2) common queue; {internal ip}
   module A1 activity (k: integer);
      ip   p1 : T1(R1) individual queue;
           p2 : T1(R2) individual queue;
           p  : T(S)   individual queue;
   end;

   body B1 for A1; {body definition} end;
   body B2 for A1; {body definition} end;
end;
```

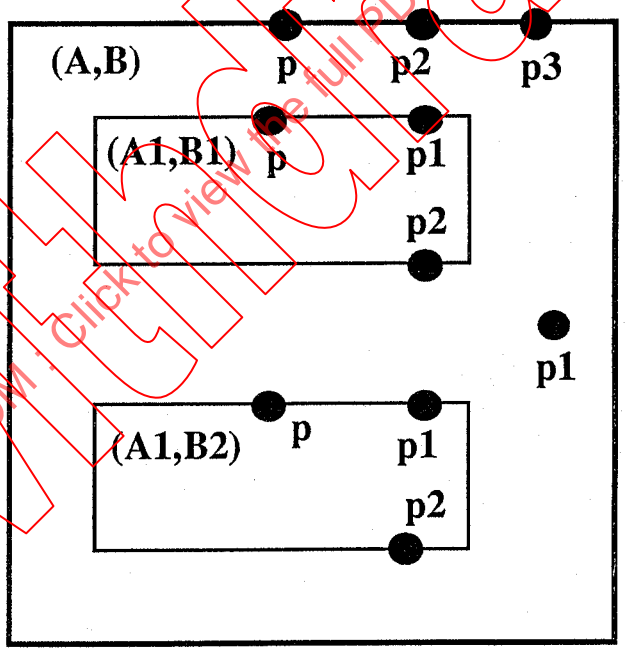

Figure A.6 - Textual hierarchy of modules

Module variables serve as references to module instances of a certain module type. For example, the declaration

```
modvar  X, Y, Z : A1
```

says that X, Y and Z are variables of the module type specified by the module header named A1.

A module instance may be created or destroyed by statements referencing module variables (**init, release** and **terminate** statements, see D.3.1 and D.3.6, respectively).

The internal behaviour of each module (instance) is defined in terms of an *extended state transition model* whose ***control* states** are defined by enumeration of their names. For example,

state IDLE, WAIT, OPEN, CLOSED

declares four control states IDLE, WAIT, OPEN and CLOSED. In other words, among the variables of a module, one implicit variable is distinguished by the keyword **state**. The state variable may assume only those values enumerated by the definition of the above form.

A *collection of control states* is sometimes referenced using a collective name which may be introduced by a **stateset** declaration. For example,

stateset IDWA = [IDLE, WAIT]

Internal interaction points may be declared to allow communication between a module and its children modules. They are declared in the same way as the external interaction points (see D.2.1), but in the declaration part of a module body definition rather then in a module header (see the declaration of the interaction point p1 within the body B for module A).

D.2.4 Initialisation part

The initialisation part of a module body, indicated by the keyword **initialise**, specifies the values of some variables of the module with which every newly created instance of this module begins its execution. In particular, local variables and the control variable state may have their values assigned. Also, some module variables may be initialised, which means that the module's children modules can be created. Creation of children modules during initialisation defines their *initial architecture*

To initialise Pascal variables, Pascal statements are used (for example, T := 5) and to initialise the state variable to, for example IDLE, we write

to IDLE

The initialisation of a module variable results in the creation of a new module instance of the variable's type. The variable is then a reference to the newly created module. To this end, the **init** statement (see D.3.1) is used. In the initialisation part, bindings may also be created between interaction points by the use of **connect** (see D.3.2) and **attach** (see D.3.4) statements. Assume the following is the initialisation part of the module (A, B) from the previous section:

```
initialise

    to IDLE
        begin
            T := 5;
            init X with B1 (0);
            init Y with B2 (1);
            init Z with B1 (4);
            connect p1 to Z.p1;
            connect X.p1 to Y.p2;
            connect Y.p1 to Z.p2;
            attach p to X.p
        end;
```

The above initialisation part creates three module instances referenced by the module variables X, Y and Z, respectively. All these instances have the same external visibility defined by the module header A1 (since the module variables X, Y and Z have been declared with module type A1). The module instances (referenced by) X and Z are both instances of the same module (A1, B1) and module instance (referenced by) Y is an instance of the module (A1, B2). The module instances X, Y and Z have been initialised with different values (respectively 0, 1 and 4) of the parameter k declared within the header of the module A1. The concrete hierarchy of module instances of figure A.7 corresponds to the hierarchical pattern of module definitions from figure A.6.

The initialisation also establishes connections and attachments between appropriate interaction points of the three newly created module instances and those of their parent module. These connections and attachments are also shown in figure 7.

The initialisation part of a module body may define more than one way of initialisation. The example below illustrates this.

```
initialise

    provided  R

      to IDLE

        begin
            T := 5;
            init X with B1 (0);
            init Y with B2 (1);
            init Z with B1 (4);
            connect p1 to Z.p1;
            connect X.p1 to Y.p2;
            connect Y.p1 to Z.p2;
            attach p to X.p
        end;

    provided  not R

      to WAIT

        begin
            T := 8;
            init X with B1 (0);
            init Y with B2 (1);
            connect X.p1 to Y.p2;
            attach p to X.p
        end;
```

The actual value of the parameter R (true or false) of the module A (see the module A header definition in D.2.2) determines how the initialisation will be done. When R is true then the module hierarchy is as in figure A.7 and when it is false as in figure A.8.

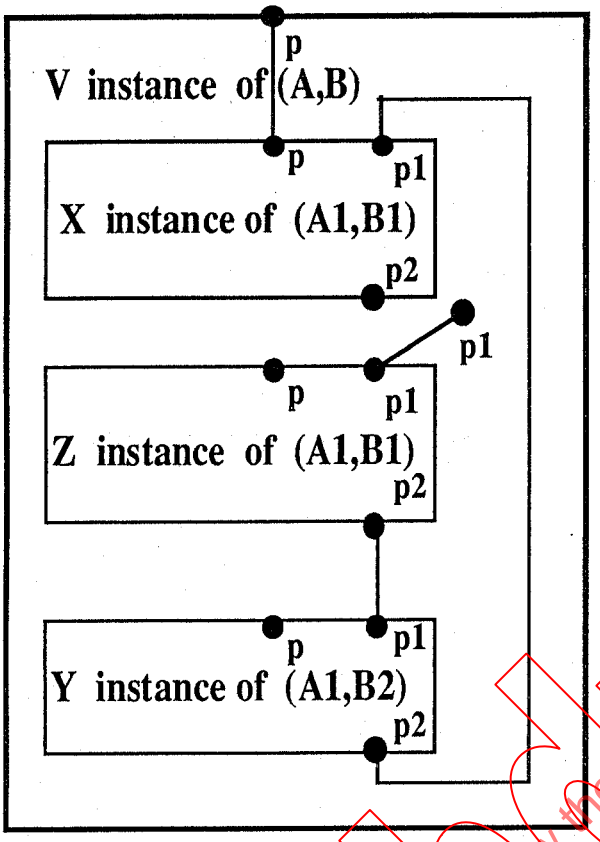

Figure A.7 - Module instance hierarchy corresponding to the textual pattern in figure A.6.

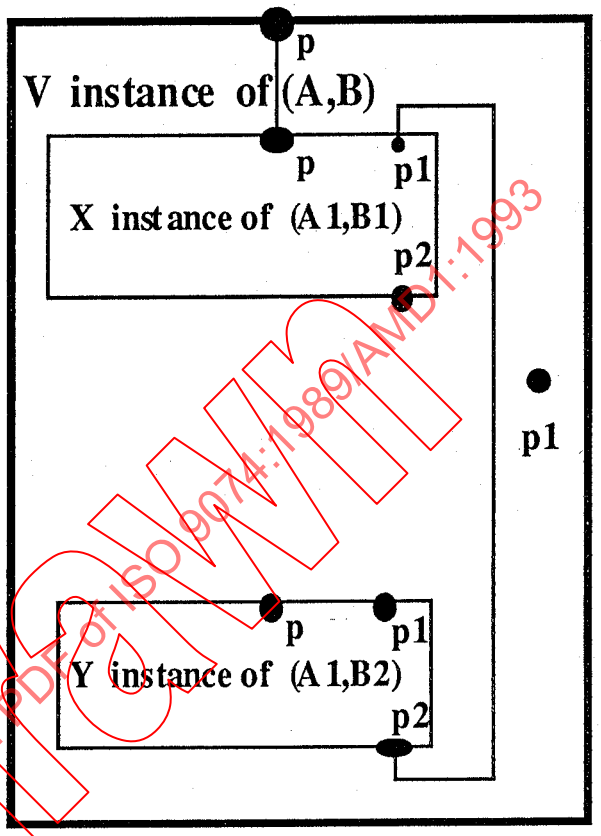

Figure A.8 - Module instance hierarchy corresponding to the textual pattern in figure A.6.

The initialisation part of a module body may also be nondeterministic. The previous example with the provided clauses removed illustrates this possibility.

It will be seen later (see D.2.5) that the text that follows the initialise keyword has the syntactical form of a transition with the restriction that the only permitted clauses are **to**-clause and the **provided**-clause. For this reason the term *initialisation transition* may also be used.

When creating module instances (executing the **init** statements) it may happen that some of them are not referenced by module variables. For example, when executing the following two statements:

init X **with** B1(0);
init X **with** B1(0);

two module instances will be created but only the second is referenced by X. There are special Estelle constructs for dealing with such non referenced instances (**forone** and **all** statements and **exist** expression - see D.3.8).

D.2.5 Transition part

The transition part describes, in detail, the internal module behaviour (see also D.1.7 and D.4.2).

The transition part is composed of a collection of transition declarations. Each transition declaration begins with the keyword **trans**. A transition may be either *expanded* or *nested*. A nested transition (see D.2.6.1) is a shorthand notation for a collection of expanded transitions. These are characterised in this section.

Each expanded transition declaration is composed of two parts :

- clause-group;
- transition-block.

Within a clause-group the following clauses define the transition *firing condition* (see also D.4.2):

- **from**-clause (**from** A1, .. ,An, where Ai (i ≥1) is a control **state** or **stateset** identifier);
- **when**-clause (**when** p.m, where p is an interaction point identifier and m an interaction identifier);
- **provided**-clause (**provided** B, where B is a boolean expression);
- **priority**-clause (**priority** n, where n is an non-negative constant);
- **delay**-clause (**delay**(E1, E2), where E1 and E2 are non-negative integer expressions).

Two other clauses may also appear within a clause-group, namely, a **to**-clause (see below) and an **any**-clause (see D.2.6.3).

Some clauses (or even all) may be omitted and at most one of each category may appear in the clause group of an expanded transition. The presence of a **when**-clause excludes a **delay**-clause and vice versa. Transitions with a **when**-clause in their conditions are called *input transitions*. Transitions without a **when**-clause are called *spontaneous*. A spontaneous transition with a **delay**-clause is called a *delayed transition*.

The *action* to be executed when transition fires is defined by:

- a **to**-clause (**to** A, where A is a control state identifier or the keyword **same**);
- a transition-block, i.e., a sequence of Estelle and Pascal statements (with specific Estelle extensions and restrictions - see D.3.9) between **begin** and **end** keywords possibly preceded by declarations of locally manipulated objects.

The **to**-clause (e.g., **to** OPEN) specifies the next control state that will be attained once the transition is fired. If omitted or specified by the keyword **same**, the next state is the same as the current state.

Example 1 below describes a transition part of a module whose detailed behaviour is explained in D.4.2. This transition part consists of six expanded transitions containing all categories of condition clauses and very simple actions. In D.2.6.1 the same transitions are joined into one equivalent nested transition.

Each expanded transition may have an associated **name**, which is introduced by an identifier. For example, each of the transitions of the example below have one of the names t1, t2, ... ,t6 associated with it.

EXAMPLE 1

```
(* transition part *)

   trans
      from IDLE
         to IDLE  (*this clause can be omitted*)
            priority medium
               when N.DATA_INDICATION
                  name t1:   begin
                                    output U.DATA_INDICATION;
                                       ak_no := ak_no + 1
                             end;
   trans
         from IDLE
            to AK_SENT
               provided (ak_no  >  0)  and  (ak_no  <=  4)
                  priority low
                     delay(min, max)
                        name t2:       begin
                                          output N.SEND_AK(ak_no)
                                       end;
   trans
         from IDLE
            to AK_SENT
               provided (ak_no > 4) and (ak_no  < 7)
                  priority high
                     delay(min)
                        name t3:       begin
                                          output N.SEND_AK(ak_no)
                                       end;
   trans
         from IDLE
            to AK_SENT
               provided ak_no = 7
                  priority high
                     name t4:          begin
                                       output N.SEND_AK(ak_no)
                                       end;
   trans
         from IDLE
            to AK_SENT
               provided ak_no = 0
                  priority low
                     delay (inactive_period)
                        name t5:       begin
                                          output N.SEND_AK(ak_no)
                                       end;
   trans
         from AK_SENT
            to IDLE
               name t6:          begin
                                    ak_no := 0
                                 end;
```

D.2.6 Shorthand's

2.6.1 Nested transitions

Expanded transitions were described in D.2.5. A nested transition is a shorthand notation for a collection of expanded transitions. Example 2 below illustrates the nesting conventions. It groups all transitions of Example 1 (D.2.5) into one nested transition (also called a transition-group).

EXAMPLE 2

```
(* transition part *)

   trans

         from IDLE

            to IDLE (*this clause cannot be omitted*)

               priority medium
                  when N.DATA_INDICATION
                     name t1:       begin
                                       output U.DATA_INDICATION;
                                       ak_no := ak_no + 1
                                    end;

            to AK_SENT

               provided (ak_no > 0) and (ak_no <= 4)
                  priority low
                     delay(min, max)
                       name t2:       begin
                                         output N.SEND_AK(ak_no)
                                      end;
               provided (ak_no > 4) and (ak_no < 7)
                  priority high
                     delay(min)
                       name t3:       begin
                                         output N.SEND_AK(ak_no)
                                      end;
               provided ak_no = 7
                  priority high
                     name t4:       begin
                                       output N.SEND_AK(ak_no)
                                    end;
               provided otherwise
                  priority low
                     delay (inactive_period)
                       name t5:       begin
                                         output N.SEND_AK(ak_no)
                                      end;

         from AK_SENT
            to IDLE
               name t6:       begin
                                 ak_no := 0
                              end;
```

Each nested transition can be transformed to a collection of expanded transitions by formal rules described in clause 7.5.2.4.1.

Algorithms to verify that nested transitions are well-formed so that they may be expanded properly, say by a compiler, are proposed and analysed in [2]. Similar algorithms are parts of existing Estelle compilers.

D.2.6.2 Provided otherwise form of the provided clause

This form of provided clause may occur only as the last provided clause of a (part of a) nested transition declaration factored by provided clauses. This means that **otherwise** refers to the boolean expressions occurring in other provided clauses of this factorisation. The **provided otherwise** form facilitates writing what could be a complex expression. Referring to the Example 2 in D.2.6.1, the **provided otherwise** form occurring within the transition named t5 is semantically equivalent to **provided** B, where B is the negation of the union of boolean expressions in the provided clauses associated with transitions t2, t3, and t4. Thus it is equivalent to

```
provided not   (((ak_no > 0) and (ak_no <= 4)) or
                  ((ak_no >4) and (ak_no < 7)) or
                  (ak_no = 7))
```

which, assuming that ak_no is of type 0..7, can be simplified to (see transition t5 in the Example 1 of D.2.5).

```
provided ak_no = 0
```

D.2.6.3 Any clause

The generic form of an **any**-clause is :

```
any domain do
```

The *domain* declares a list of local variables of finite ordinal types.

The **any**-clause is one of the clauses that may be used within (a clause-group of) a transition declaration. It indicates a macro-like expansion of the remainder of the transition declaration that follows it.

For example a transition declaration :

```
trans
    from S1 to S2
      any  n : 1..2;  k : 3..4  do
        when p[n].m
          begin
            variable := k
          end;
```

is a shorthand for the following nested transition :

```
trans

    from S1 to S2
      when p[1].m
        begin
            variable := 3
        end;

      when p[1].m
        begin
            variable := 4
        end;

      when p[2].m
        begin
            variable := 3
        end;

      when p[2].m
        begin
            variable := 4
        end;
```

D.3 Estelle statements

D.3.1 Init statement

The init-statement is used to create module instances. These module instances are children of the module that creates them (i.e., the module that issues the init-statement).

The two generic forms of the init-statement are

init module-variable **with** body-identifier;

init module-variable **with** body-identifier (actual-module-parameter-list).

The only difference between the two forms above is that the second one allows parameters to be passed (as in a Pascal procedure) to the module instance being created. These parameters should be passed when the module header definition specifies such parameters (see D.2.2).

The *module-variable* in the **init**-statement refers to the newly created module instance. This module-variable has to be previously declared as being of the module-type (module-header identifier) with which the module body indicated by *body-identifier* is associated (see D.2.2). Thus, the module instance created is of this type and may be referenced by the module-variable. There may be several module-bodies that could be associated with a module-header (see D.2.2) and the **init**-statement serves to select one of them.

If a module-variable, say X, is re-used (within another **init**-statement) or assigned (e.g. X := Y), the instance it is referring to changes, but the previously referenced module instance does not cease to exist.

For example, assume that the module-variable X is declared to be of type A (i.e., X : A), then,

init X **with** B; **init** X **with** B

creates two module instances both of the type defined by the module-header identifier A (both with the same external visibility's), and with identical internal behaviour defined by the body B.

Note, however, that only the instance created second is referenced by the module-variable X. The first one is not referenced by any module-variable (is not referenced at all from the specification point of view), and the only way to access it is through the use of the **forone** or **all** statements or of the **exist** expression (see D.3.8).

The **init**-statement may be used within an **initialisation**-part (see D.2.4) or **transition**-part (see D.2.5) of a module body declaration allowing, respectively, the static or dynamic creation of children module instances of a given module.

D.3.2 Connect statement

The generic forms of a connect-statement are

connect internal-ip **to** child-external-ip;

connect child-external-ip **to** internal-ip;

connect child-external-ip **to** child-external-ip;

connect internal-ip **to** internal-ip.

A **connect** statement issued by a module is used to connect:

1) an internal interaction point of the module to an external interaction point of its child module (the first two forms above);

2) external interaction points of two children modules of the module (the third form above);

3) two internal interaction points of the module (the forth form above).

Recall that an interaction point is external if it is declared as part of a module's header; an interaction point is internal if it is declared within a module's body.

The two interaction points referenced by a connect statement must refer, in their declarations, to the same channel and they must play opposite roles (must have opposite types - see D.2.1) with respect to this channel.

Observe that a connect statement referencing an external interaction point of the module issuing the statement is syntactically invalid; Thus, the situation depicted in figure A.9a is impossible.

A connect statement has no effect when it attempts to connect an interaction point that is currently already connected (or attached by the module issuing the connect statement - see D.3.4) or when it attempts to connect an interaction point to itself; the situations depicted in figure A.9b, c and figure A.11c, d are thus impossible.

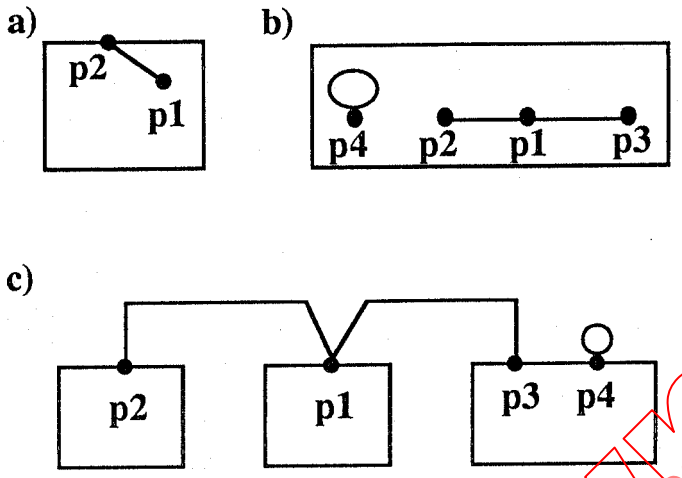

Figure A.9 - Impossible configurations of interaction point interconnections

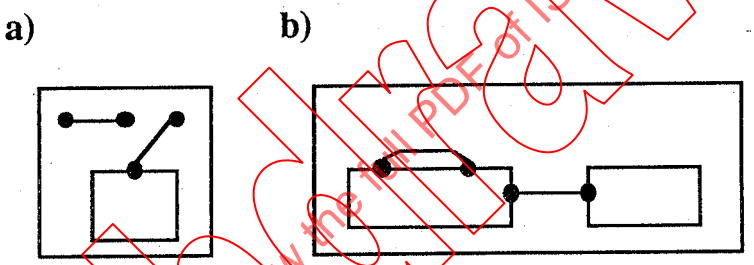

Figure A.10 - Possible configurations of interaction point interconnections

In summary, at a given moment

(1) an internal interaction point of a module may be connected only to another internal interaction point of the module or to an external interaction point of a child module;

(2) an interaction point may be connected to at most one interaction point, and it cannot be connected to itself.

Note also that connecting two interaction points does not have any influence on the contents of queues associated with these interaction points.

D.3.3 Disconnect statement

The generic forms of a disconnect statement are

disconnect internal-ip;

disconnect child-external-ip;

disconnect module-variable.

A disconnect statement in its first two forms disconnects a pair of interaction points even though only one of them is explicitly specified by the statement; the pair is implicit because the interaction point explicitly specified is supposed to be currently connected to another one as a result of some previously executed connect statement (otherwise the disconnect statement has no effect).

A disconnect statement in its last form is a generalisation of the second form in that it applies to all the external interaction points of a child (indicated by the module variable) of the module issuing the disconnect statement.

A disconnect statement does not have any influence on the contents of queues associated with the interaction points being disconnected.

D.3.4 Attach statement

The generic form of an attach-statement is

attach external-ip **to** child-external-ip.

The sequence of actions of the attach statement are as follows:

1) attach (bind) the pair of interaction points specified, the first of them must be an external interaction point of the module issuing the statement (identified by the interaction point reference) and the second must be an external interaction point of one of the module's children (identified by the interaction point reference prefixed by a module variable referencing the child);

2) remove from the queue associated with the first interaction point all those interactions that have been previously enqueued through this first interaction point; recall that the queue associated to this interaction point may be shared ("common queue") with other interaction points through which interactions could have been also enqueued;

3) append the interactions just removed to the current contents of the queue associated with:

   a) the second (child) interaction point specified, if it is not in turn attached to another child interaction point (i.e., the second interaction point is the end-point), otherwise,

   b) the interaction point that is the end-point of the chain of attachments whose segments attach the consecutive pairs of interaction points beginning with the second child interaction points and ending with the interaction point of a great .. great grand child of the child.

The two interaction points identified in an attach statement must refer, in their declarations, to the same channel and they must play the same roles (must have the same type - see D.2.1) with respect to this channel.

An attach statement has no effect in the following cases:

a) when an attempt is made to attach an interaction point of the module issuing the attach statement that is already attached to another interaction point of one of the module's children; the situation depicted in figure A.11a is thus impossible;

b) when an attempt is made to attach an interaction point of the module issuing the attach statement to an interaction point of one of the module's children that is:

   - already attached to the module's interaction point, or
   - already connected to an interaction point.

Thus the situations depicted in figure A.11b, 11c and 11d are impossible.

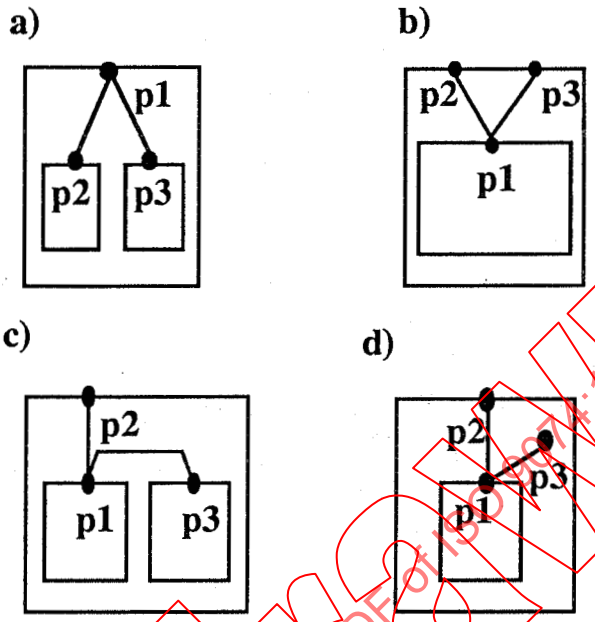

Figure A.11 - Impossible configurations of interaction point attachments and connections

Note, however that the situations depicted in figure A.12a and 12b are perfectly valid.

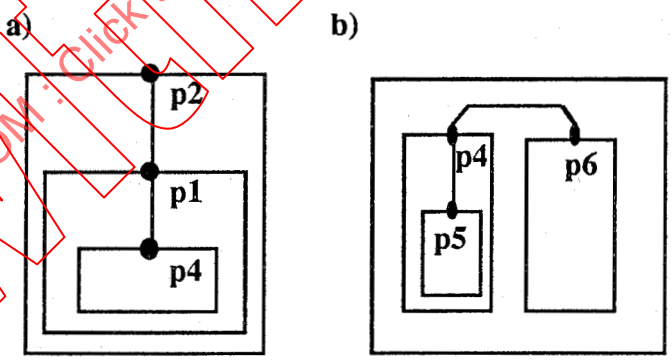

Figure A.12 - Possible configurations of interaction point attachments and connections

In summary, at a given moment,

(1) an external interaction point of a module may be attached to at most one external interaction point of its parent module and to at most one external interaction point of its children;

(2) an external interaction point of a module attached to an external interaction point of its parent module cannot be simultaneously connected.

D.3.5 Detach statement

The generic forms of a detach statement are:

detach external-ip;

detach child-external-ip;

detach module-variable.

A detach statement in the first two forms detaches a pair of interaction points even though only one of them is explicitly specified by the statement; the pair is implicit because the interaction point explicitly specified is supposed to be attached, at that moment, to the implied one as a result of some previously executed attach statement (otherwise the detach statement has no effect).

A detach statement in its last form is a generalisation of the second form in that it applies to all the external interaction points of a child (indicated by the module-variable) of the module issuing the detach statement.

Besides detaching the pair(s) of interaction points, the detach statement also influences the content of some queues associated with the interaction-points. To explain this point let us consider first the case where the child's external interaction point (explicitly or implicitly referenced by the detach statement) is not in turn, at that moment, attached to its child's external interaction point (i.e., it is an end-point). In such a case, all interactions in the queue associated with the child's external interaction point that were enqueued through this interaction point while it was attached are removed and appended to the queue associated with the external interaction point of the module issuing the detach statement.

In general, a detach statement issued by a module may break a chain of attachments going from the child's external interaction point of the module towards the descendant modules. In such a case all the interactions from the queue associated with the interaction point that is the end-point of the chain of attachments (all the way down to the great, great.... great grand child) that were enqueued through this interaction point and that passed via the external interaction point of the module issuing the detach statement, while it was chain-attached, are removed and appended to the external interaction point of the module issuing the detach statement.

D.3.6 Release and terminate statements

The generic forms of the release and terminate statements are as follows:

release module-variable;

terminate module-variable.

The result of the **release** X statement, where X is the module-variable identifying a child of the module instance issuing the statement, is equivalent to the following sequence of actions:

(1) executing, in any order, the **disconnect** X and **detach** X statements;

(2) destroying the module instance identified by the module-variable X as well as all its descendant instances; the value of X as well as any other module-variables identifying the module instance X (say Z if an assignment Z : = X had been previously made) becomes undefined, as if it had never been initialised.

The result of a **terminate**-statement differs from the result of a **release**-statement in that the execution of the implicit **detach** X statement within the execution of the release statement is restricted only to detach all external interaction points of the module instance identified by the module-variable X without moving the

interactions in queues. This restricted detach is called *simple-detach* in the formal semantic definition (see 9.6.6.2.4).

In other words, **terminate** X statement abruptly destroys the module instance X (and all its descendants) including all enqueued interactions and bindings of interaction points.

Note that since the module instance identified by the module-variable references a child of the module instance issuing the **release** or **terminate** statement, these statements can be used only to destroy the child and all its descendants; a module may not destroy itself or a sibling module.

Note also that the execution of the sequence of statements

detach X; **terminate** X

gives the same result as executing **release** X.

D.3.7 Output statement

The generic forms of the output statement are as follows:

output ip-reference.interaction-identifier;

output ip-reference.interaction-identifier(actual-parameter-list).

By using an **output**-statement a module sends an interaction (possibly with actual parameter values) via a specified interaction point. For example, the statement

output p1.REQUEST(3, true)

sends the interaction REQUEST(3, true) via the interaction point p1.

If p1 and p2 are the two end-points of a communication link (see D.1.3.1), then the statement **output** p1.m appends interaction m to the queue associated with the interaction point p2.

Since an interaction end-point may be linked to at most one other interaction end-point, there is a unique receiver (if any) of the interaction sent by a module.

When p1 is not the end-point of a communication link, the statement **output** p1.m has no effect (the interaction m is considered to be lost).

D.3.8 All and forone statements, exist expression

In Pascal and Estelle, all objects of a given type comprise a domain. Estelle provides repetition and selector statements, and a boolean operator, which operate over a named domain (a variable declared either as a module or as an ordinal type). The repetition statement is named **all**, the selector statement is named **forone,** and the boolean operator is named **exist**.

D.3.8.1 All statement

The generic form of the **all** statement is:

all domain **do** statement;

The statement contained in the **do** part of the **all** statement can be a compound statement, comprising several statements between the Pascal keywords **begin** and **end**.

The *domain* indicates the set over which the search is to be performed, that is, either a collection of module instances of the same module-type, or one or more finite ordinal types.

When the domain of the **all** statement is empty the **all** statement has no effect.

When **all** is used with module instances, the domain declares a module variable and is restricted to one such variable and one module-header-identifier.

Since a module-header-identifier identifies the module-type, the module instances to be examined and manipulated within an **all** statement are declared as follows

module-variable-identifier : module-header-identifier

This has the effect of declaring a module-variable local to the scope of the **all** statement; the module-variable is of the module-type identified by the module-header-identifier.

For example, during a reset operation, a parent managing several children, each representing a connection, might use

```
all M : ModuleType do M.InProgress := false
```

to set an exported variable named "InProgress" to false for each child module of type "ModuleType".

Assume that a parent has initialised several modules whose type is "network". Also assume that because of a catastrophic failure, it is necessary to release all instances of the network modules. Then it is possible to specify:

```
trans
  when system failure indication
    priority highest
      to closed
        begin
          all net : network do
          release net
        end;
```

All module instances of the network module type are released. Since we have assumed the existence of several such instances, supporting separate subnetworks, all of them will be released. First "net" is assigned the value of one (arbitrary) of the module instance references of the type "network" and it is released; then the operation is repeated for all the others. Note that the order in which module instances are released is not specified.

Note also that module instances manipulated within an **all** statement are not necessarily referenced by module variables declared outside the all statement domain (see D.3.1).

As mentioned above, the domain of an **all** statement may also deal with variables of finite ordinal types. In this case the domain of the **all** statement may be specified as follows

variable-list 1 : type-denoter 1;.....variable-list N : type-denoter N

All these variables have scope local to the **all** statement

If the domain of the **all** statement contains more than one variable, the statement following the **do** keyword is executed for each instance of an object in each domain specified.

In the above example, releasing "network" modules means that the parent module receives the contents of all queues associated with all interaction points of the released modules that have been attached to its interaction points. Suppose that this is not desirable, since the only queue contents the parent is interested in are those associated with the interaction points grouped by "colours" and "sorts" by means of the following interaction point declaration (within the "network" module-header declaration):

p : **array** [ip_colour, ip_sort] **of** channel1 (R1)

where the types ip_colour and ip_sort have been defined earlier by:

```
type
        ip_colour = (red, green, blue);
        ip_sort  = 1..3;
```

Note, however, that the parent has also other interaction points attached to children "network" modules' interaction points other then those grouped within the declaration of p.

One could now reformulate the transition from the previous example into:

```
trans
   when system.failure_indication
      priority highest
         to closed
            begin
               all net: network do
               all colour: ip_colour; sort: ip_sort do
                  begin
                     detach net.p[colour, sort];
                  end;
               terminate net
            end;
```

In the example above, the interior **all** statement iterates over two domains named ip_colour and ip_sort. The order of iteration is nondeterministic. Therefore, the "colour" variable evaluates to red, green and blue while the "sort" variable evaluates to the values 1, 2 and 3; but unlike nested Pascal "for" loops, the order of evaluation is not known.

D.3.8.2 Forone statement

The **forone** operation complements the **all** operation. It is used to search for an object satisfying given criteria, which may be expressed as a boolean expression. It has two generic forms :

forone domain **suchthat** boolean-expression **do** statement;

forone domain **suchthat** boolean-expression **do** statement 1 **otherwise** statement 2

where the *domain* is declared in the same way as in the **all**-statement.

The boolean-expression is evaluated for each instance of an object in the domain(s) identified until it yields the value true. When an object is encountered for which the boolean-expression is true, statement 1

following the keyword **do** is executed and the operation terminates. If the boolean-expression is false for all instances of objects in the domain (or the domain is empty), then (if the second form was used) the statement 2 that follows the keyword **otherwise** is executed.

The boolean-expression may contain, for example, exported variables (shared between parent and child) that help identify the module sought.

Assume for the following example that a "network" module may handle only one connection at a time through its interaction point NET_SAP; also assume that a boolean variable "is_busy" is exported by each instance of the network module to indicate whether or not the module is currently attached to its parent.

A "connection_request" interaction may arrive any time from a user to the parent module (of the network modules) through an interaction point user[k] for any k = 1,..,N. If it happens, and if a network module instance exists which is not busy (is_busy = false), then communication with this module instance may be established (i.e., the parent attaches its user[k] interaction point to the NET_SAP interaction point of the network module instance). If all existing network module instances are busy or if such instances do not exist at all, then a new network module instance must be created first and then attached to the parent.

This could be expressed by the following transition (declared within the parent module) where new_net is a previously declared module variable of the "network" type:

```
trans

  any k : 1..N do
    when user[k].connecton_request
      begin
        forone net : network
          suchthat not net.is_busy do
            begin
              attach user[k] to net.NET_SAP;
              net.is_busy := true
            end;
          otherwise
            begin
              init new_net with network_body;
              attach user[k] to new_net.NET_SAP;
              new_net.is_busy := true
            end;
      end;
```


D.3.8.3 Exist expression

Exist expression provides the facility to determine if an instance of an object exists; it is a relational expression returning either the value true or false. As such, it may be used as a factor in a more complex expression. Its form is:

exist domain **suchthat** boolean-expression;

where the *domain* is declared in the same way as in the **all** and **forone** statements.

Assume that in the previous example (see the end of D.3.8.2) the network modules could handle more than one connection. For example, the interaction point (within the network module-header) could have been declared as:

NET_SAP : **array**[1..M] **of** Net_channel(R1);

Then the value of the variable "is_busy" could have been determined by the **exist** expression (within a more complex **forone** statement):

exist i : 1..M **suchthat** is_attached_NET_SAP(i)

where "is_attached_NET_SAP(i)" is a function returning true or false depending whether or not the interaction point NET_SAP[i] is attached to an interaction point of a parent.

D.3.9 Pascal restrictions in Estelle

Estelle makes use of Pascal [41] in the transition blocks to specify actions that take place during the transitions. Only level 0 Pascal is used, thus excluding the use of conformant arrays. Because Estelle is a specification technique rather than a programming language, some restrictions were introduced to constrain the Pascal components. Integers and real numbers have their usual mathematical meaning in Estelle, so implementation dependent constraints such as MAXINT and the precision of real numbers do not enter into Estelle. Similarly, all those features of Pascal that relate to file manipulation have been removed (**file, text, get, put, read, write, readln, writeln, eof, eoln**). Also the keyword **program** has been removed.

The use of **goto** statements and **labels** has been severely restricted, so that a **goto** acts like a "return" statement. A **goto** may be used only within a procedure or a function, never directly within a transition block. Consider the case where a procedure A invokes procedure B. In Pascal, a **goto** in procedure B could transfer control to any **label** in procedure A or procedure B. In Estelle, however, it may transfer control only to the end of procedure B.

No function in Estelle may have side-effects, so there are no complications introduced by the order of evaluation of expressions or by evaluation of the **provided** clause of transitions or by evaluation of actual parameters of functions, procedures, or modules. Side-effects are avoided by requiring that functions be *demonstrably pure* (as defined in 8.2.5.1). The key idea of that definition is that a demonstrably pure function cannot modify either directly or indirectly (through a pointer or by calling another procedure or function) any variable that is not local. In addition to requiring all functions to be demonstrably pure, Estelle permits procedures to be declared **pure**, meaning that they must be demonstrably pure.

As a convenience, functions are permitted to return arbitrary types, but the syntax does not permit complex types returned by functions to be used within arithmetic or logical expressions. Nevertheless, the returned value can be used as the right-hand-side of an assignment.

Estelle makes it possible to specify that the definition of a procedure, a function, or a module is to be found elsewhere. This is indicated by the directive **external**, which is thus simply a way to indicate necessary text substitution. This is in contrast to the directive **primitive**, which indicates that the description of a procedure or function (but never a module) is not given in Estelle. This is frequently used with data types indicated as "..." as a way of deferring specification or of leaving implementation details to implementors. As an example, buffers could be specified by:

type BufferType = ...;

and the routines to manipulate buffers (e.g., Insert, Extract, IsEmpty, etc.) could then be declared **primitive**. Note that **primitive** functions and procedures (and only such) have global scope.

Pointers are a necessary evil of Pascal, but their use is restricted within Estelle. Pointers (and any variable containing pointers as components) are excluded from use as parameters of interactions and as parameters

and exported variables of modules. As noted above, they are also excluded from use as parameters of functions and **pure** procedures.

Finally we mention here that Estelle constructs cannot be used in procedures or functions. Technically this is not a restriction to Pascal, because these construct were never a part of Pascal. Since the additional Estelle statements cannot be hidden in procedures or functions, they are easily seen. This is important, as they may specify the information about module structure and inter-module communication.

D.4 Behavior of Estelle specifications

D.4.1 Specification module

All modules defined as described in the preceding sections are textually embodied in a main module called **specification** module. This unique module is defined as follows :

```
specification SPEC-NAME [system-class];
   [default-option]
   [time-option]
      body definition
end.
```

where the system-class attribute (if any) is either **systemprocess** or **systemactivity**, and the default-option is either **individual queue** or **common queue** (the parts in square brackets are optional).

The intent behind defining common or individual queue in a specification module definition is to give the default assignment of queues to those interaction points of the whole specification for which this assignment is omitted in their declarations.

The time-option indicates the unit of time (millisecond, second, etc.) applicable to the specification. A non-negative integer expression within a **delay**-clause indicates the number of units the execution of a transition must (or may) be delayed (see the last example in D.4.2 for interpretation of delay-clauses).

The above specification definition is considered semantically equivalent to the following module definition (module header and module body declarations):

```
module ANY-NAME [system-class];
end;

body SPEC_NAME for ANY-NAME;
     body definition
end;
```

where ANY_NAME may be chosen arbitrarily and body definition takes into account the default-option.

Note that the specification module has neither interaction points nor exported variables. This means that an Estelle specification is not itself a module that communicates with other external modules. In practice, a specification body often constitutes a general *framework* for a system being defined, i.e., it provides a global context necessary for the system definition and initialisation. Note that, the definitions of constants, types, channels and procedures or functions declared as **primitive** are visible by any descendant module.

It is assumed that there exists only one instance of a specification module (main instance).

The example below illustrates this "framework" role of the specification module which serves (in this example) to define only an overall system architecture including modules' interfaces (so-called *high-level-design*) without going into details of its components' definitions by leaving their bodies unspecified, i.e., **external**.

```
specification EXAMPLE;

   default individual queue;
   timescale second;

   channel UCH(User,Provider);
      by Provider: DATA_INDICATION;

   channel NCH(User,Provider);
      by User: DATA_INDICATION;
      by Provider: SEND_AK(x: integer);

   module USER systemactivity;
      ip U: UCH(User);
   end;

   body USER_BODY for USER; external;

   module RECEIVER systemactivity;
      ip U: UCH(Provider); N: NCH(Provider);
   end;

   body RECEIVER_BODY for RECEIVER; external; (* see D.4.2 *)

   module NETWORK systemactivity;
      ip N: NCH(User);
   end;

   body NETWORK_BODY for NETWORK; external;

   modvar X: USER; Y: RECEIVER; Z: NETWORK;

   initialize
      begin
         init X with USER_BODY;
         init Y with RECEIVER_BODY;
         init Z with NETWORK_BODY;
         connect X.U to Y.U;
         connect Y.N to Z.N;
      end;

end.
```

The specification declares and initialises three systems X, Y and Z (i.e., the systems are referenced by module variables X, Y and Z of types USER, RECEIVER and NETWORK, respectively). These systems exchange some messages through their interaction points connected as declared in the initialisation part of the specification. While the RECEIVER_BODY is further specified in D.4.2 as one simple module body

(without substructuring), the other systems remain "external" and their definitions can be substructured differently in subsequent refinements of the specification.

The graphical representation of the specification EXAMPLE is presented in figure A.13.

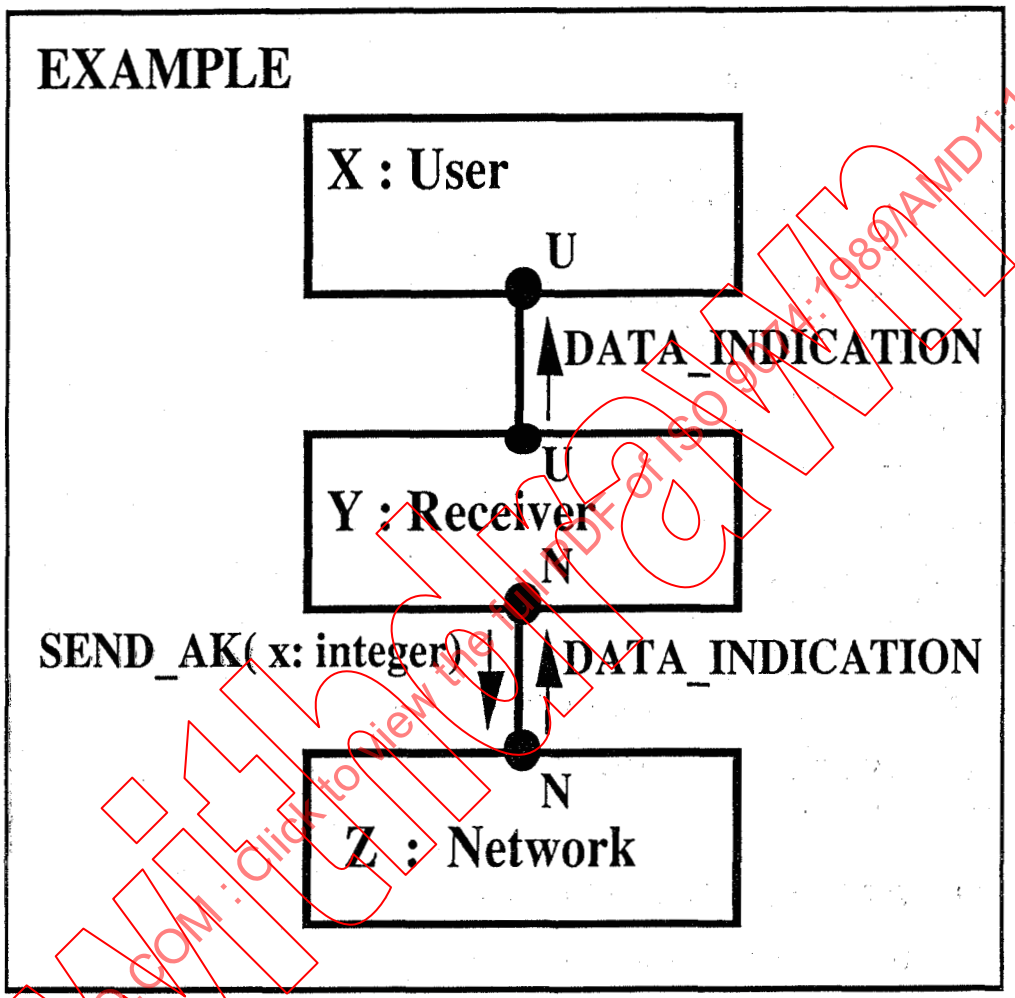

Figure A.13 - Graphical representation of the specification EXAMPLE (high-level-design)

D.4.2 Internal behaviour of a module

As noted earlier (see D.1.7) the behaviour of a module is expressed in terms of an extended state transition model in which one computation step is defined by execution of a transition action in the module internal state. The criteria expressed by condition clauses of a transition determine whether the transition is *firable* (or *ready-to-fire*) in a state of the module (and at a given moment of time, if it concerns a delayed transition). The action of one of those firable transitions eventually executes and the module will reach a new state.

Note that, the term *state* means here a complex structure composed of many components such as: value of the control state, values of variables, contents of FIFO queues associated with interaction points and the status of the module internal structure (submodule instances, bindings between interaction points, etc.).

This section defines when a transition is firable and it explains through simple examples how to represent a desired behaviour in Estelle.

A **from**-clause is said to be *satisfied* in a module state if the current value of the module's control state is among those listed by the **from**-clause. For example, if IDLE is the current control state of a module instance, then all three of the following **from**-clauses are satisfied (assume that IDWA=[IDLE,WAIT]):

from IDLE,
from IDLE, OPEN, CLOSE,
from IDWA,

The **when** p.m clause is *satisfied* in a module state if the interaction m is at the head of the queue associated with the interaction point p.

The **provided** B clause is *satisfied* in a module state if the boolean expression B evaluates to "true" in that state.

A transition is said to be *enabled* in a module state if the **from**, **when** and **provided** clauses, if present in the clausegroup of the transition, are all satisfied in this state.

A transition is said to be *firable* (or *ready-to-fire*) in a module state and at a given moment of time if

(a) it is *enabled* in the state, and if it is a delayed transition, with its delay clause "**delay**(E1,E2)", then it must have remained enabled for at least E1 time units, and

(b) it has the highest priority among transitions satisfying (a), where "higher priority" corresponds to "smaller nonnegative integer".

With the above definitions in mind, the first two examples below illustrate how to represent in Estelle a simple finite state automaton (FSA).

The first of them (specification Example1) is a direct translation into Estelle of transitions of the FSA given by the diagram in figure A.14b and state table in figure A.14c, while the second (specification Example2) defines an equivalent behaviour (i.e., that in figure A.14e and figure A.14f) in a more concise way due to the introduced auxiliary variable "x" and interactions parameter "p". In both cases the description consists of one Estelle module body E1 with its header E, respectively (the interface is, however, different - see **channel** definitions). The module is embedded into a "framework" specification module which, together with the module header E, serves to declare the required interface with the environment as given in Figures 14a and 14e. In the second case, the type T (of the interaction parameter "p" and of the variable x) is also defined within this "framework".

```
specification Example1;

    default individual queue;
      channel U(R1,R2);
        by R1: put;
      channel S(R1,R2);
        by R1: dt0; dt1;
        by R2: ak0; ak1;
      module E systemprocess;
        ip U: U(R2);   S: S(R1);
      end;
      body E1 for E;
        state s0, s1, s2, s3;
        initialize to s0 begin end;
        trans          when U.put
                       from s0 to s1
                          begin output S.dt0 end;
                       from s2 to s3
                          begin output S.dt1 end;
                    when S.ak0
                       from s1 to s2
                          begin end;
                    when S.ak1
                       from s3 to s0
                          begin end;
    end;
end.


specification Example2;

    default individual queue;
      type T = 0..1;
      channel U(R1,R2);
        by R1: put;
      channel S(R1,R2);
        by R1: dt(p:T);
        by R2: ak(p:T);
      module E systemprocess;
        ip U: U(R2);   S: S(R1);
      end;
      body E1 for E;
        state S0, S1;
        var x: T;
        initialize to S0 begin x := 0 end;
        trans          when U.put
                       from S0 to S1
                          begin output S.dt(x) end;
                    when S.ak(p)
                       provided p=x
                          from S1 to S0
                             begin x := 1-x end;
    end;
end.
```

Note that in the second description two extensions have been made. The first is the parameter "p" of the enumerated type T = 0..1 which permits the declaration of two-element classes "dt" and "ak" of interactions instead of declaring their elements as four separate interactions (observe that "dt(0)" corresponds to "dt0" in Example1, etc.). The second is the variable "x" of the same type T. It permits, in a similar way, the replacement of the four previous control states by only two (observe that in Example2, the situation of being in the control state "S0" with x=0 , corresponds to the situation of being in the control state "s0" in Example1, etc.).

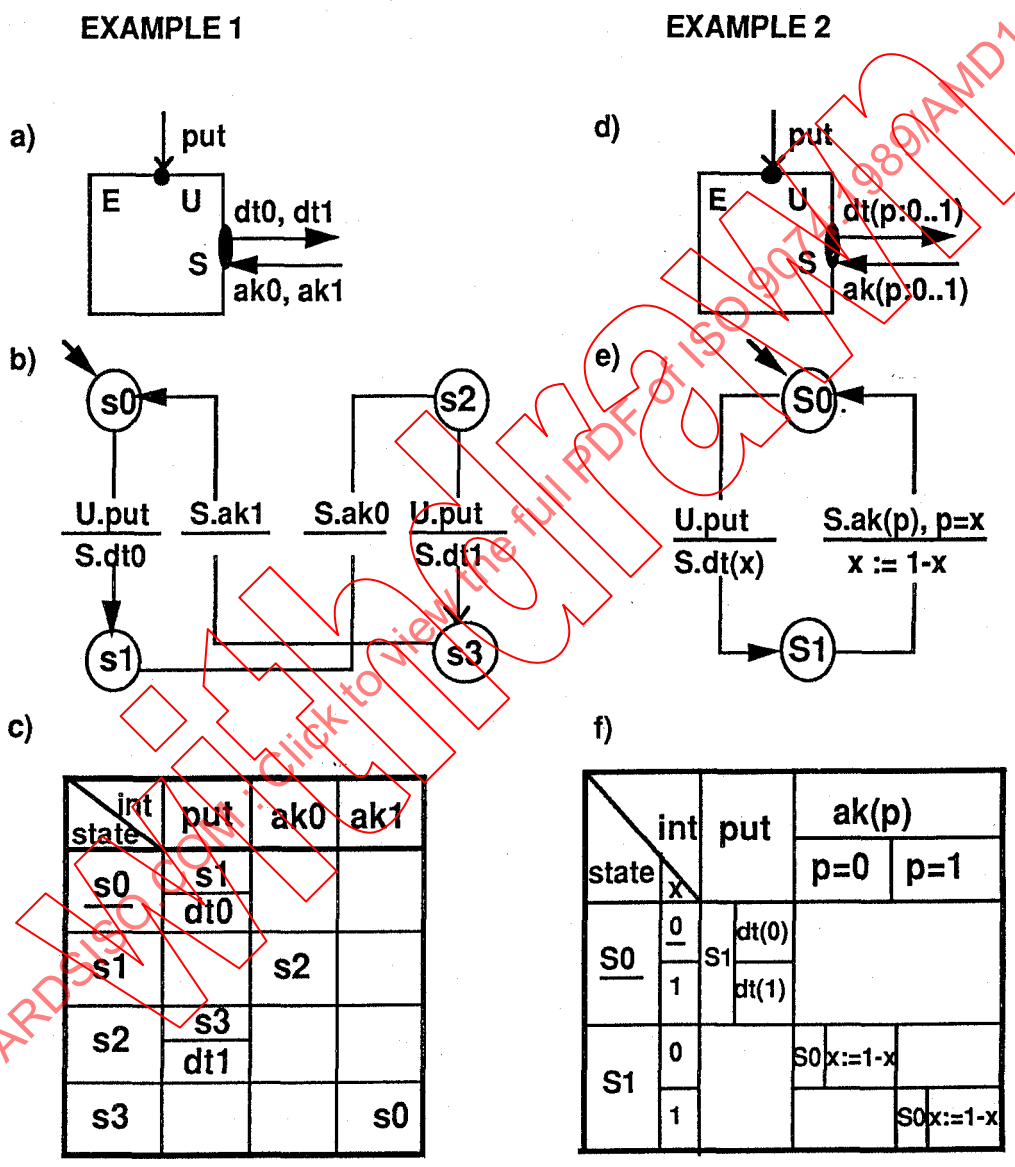

Figure A.14 - Graphical representation of interfaces of modules E ((a) and (d)) and of their internal behaviour represented by state graphs ((b) and (e)) and state tables ((c) and (f), respectively, for Example1 and Example2 specifications

The next example specifies in Estelle a behaviour in which a combination of all categories of condition clauses is used to satisfy the requirements formulated informally below. More specifically this example illustrates the use and interpretation of delay and priority clauses.

As observed earlier the computational model for Estelle is formulated as far as possible in time-independent terms. However, some Estelle spontaneous transitions may contain a **delay**-clause of the form "**delay**(E1,E2)". The intent of this clause is to indicate that execution of the transition (if it is enabled) must be delayed. The minimum time the transition must be delayed and the maximum time it may be delayed are initially specified by the values of integer expressions E1 and E2 respectively. An implementation may choose a concrete delay value in the closed interval determined by these expressions. Let us note that "**delay**(E1)" means the same as "**delay**(E1,E1)" and that **delay**(E1,*) means that the maximum delay time is *infinity*.

The body definition which we describe in this example can replace the "external" body parameter of the RECEIVER module of a simple communication scheme defined in D.4.1. The necessary channel definitions and declaration of interaction points are then defined at the specification level of the EXAMPLE in D.4.1. The transition part of this module is that from Example 2 in 2.6.1 (and equivalent to that of D.2.5).

The following narrative describes the behaviour of a protocol which assumes that:

- multiple protocol data units (messages) may be acknowledged in a single acknowledgement;
- each protocol data unit shall be acknowledged after some maximum time (max);
- it is desirable to acknowledge each protocol data unit received after a minimum time (min), but when more than four are received an acknowledgement must be sent after this minimum delay (min);
- the maximum number of unacknowledged protocol data units is seven;
- when the system remains inactive too long (inactive period), "dummy" acknowledgements are generated to provide minimal activity to prevent disconnection.

The following module body describes a solution in Estelle. The specification in D.4.1 assumes a time scale resolution of 1 second (see the "timescale" option in the specification).

```
body RECEIVER_BODY for RECEIVER;

    (* declaration part *)
       type        time_period = integer;
       const          high = 0;
                   medium = 1;
                   low = 2;
       state          IDLE, AK_SENT;
       var            ak_no : 0..7;
                   min, max, inactive_period : time_period;

    (* initialisation part *)
       initialize
          to IDLE
             begin
                min := 1;
                max := 20;
                inactive_period := 60;
                ak_no := 0;
             end;
```

```
(* transition part *)
   trans  (* transition part from Example of D.2.6.1 *)
      from IDLE
         to IDLE
            priority medium
               when N.DATA_INDICATION
                  name t1:      begin
                                   output U.DATA_INDICATION;
                                   ak_no := ak_no + 1
                                end;
         to AK_SENT
            provided (ak_no  > 0) and (ak_no  <= 4)
               priority low
                  delay(min,max)
                     name t2:      begin
                                      output N.SEND_AK(ak_no)
                                   end;
            provided (ak_no  >4) and (ak_no  <7)
               priority high
                  delay(min)
                     name t3:      begin
                                      output N.SEND_AK(ak_no)
                                   end;
            provided ak_no = 7
               priority high
                  name t4:      begin
                                   output N.SEND_AK(ak_no)
                                end;
            provided  otherwise
               priority low
                  delay (inactive_period)
                     name t5:      begin
                                      output N.SEND_AK(ak_no)
                                   end;
      from AK_SENT
         to IDLE
            name t6:      begin
                             ak_no := 0
                          end;

end;  {of body RECEIVER_BODY for RECEIVER}
```

As explained in D.2.6.1 the transition part of the RECEIVER_BODY consists of one nested transition representing in fact six expanded transitions. The outer-most level of factorisation (**from**-clauses) divides these six transitions into two groups: t1-t5 and t6. **To**-clauses separate the first transition and the nested transition which groups expanded transitions t2-t5. These transitions in turn, are factored by their **provided**-clauses.

From the initialisation part (see D.4.1), we see that the module begins in its IDLE control state, with the delay values appropriately initialised and the number of messages that remain to be acknowledged set to 0. Note that immediately after this initialisation, the delay-timer associated with transition t5 begins to run since the transition becomes enabled.

If a DATA_INDICATION message arrives during the inactive period (60 seconds), then the message is transmitted to USER and the number of messages to acknowledge increases by 1 (transition t1). In this

case, the timer of transition t5 is cancelled. Otherwise, i.e., when 60 seconds passes without any incoming message, an "artificial" acknowledgement is sent to prevent disconnection.

Note that immediately after the first message arrived and was served, the delay-timer of transition t2 starts to run, but the transition may be fired only if: there is no new message to be served (otherwise transition t1 would fire), 1 second already has passed, and the number of messages to be acknowledged is not greater than 4. This transition certainly will be fired if the above situation has remained unchanged for 20 seconds. If the transition was fired or more than 4 messages have been served in between, then the transition is no longer enabled and its timer is cancelled.

Immediately after the 5th consecutive message is served (transition t1) but before all five have been acknowledged, the delay-timer of transition t3 is turned on. The timer will run exactly 1 second and the transition will be executed if the number of unacknowledged messages during this time remains less than 7. Otherwise, if the 7th message is served (transition t1) during this period, transition t3 will not be executed and its timer will be cancelled; the transition t4 will be executed instead.

It is worth noting the role that is played by priorities assigned to transitions. Due to them, in a condition of heavy traffic of arriving messages (more than 7 in less than one second), acknowledgements are always sent for blocks of 7 messages (the priority of transition t4 is higher than that of transition t1). In less heavy traffic (more than 4 but less than 7 messages in one second), the acknowledgements are sent always for blocks of 5-6 messages. If the messages arrive at a rate of 1 to 4 every 20 seconds, they may be acknowledged one-by-one or in blocks of 2-4 messages depending on the message distribution in that time and the real delay value in the time interval <min, max > (the input transition t1 has a higher priority than transition t2). If no more than one message arrives every 20 seconds, the messages certainly (and in any implementation) will be acknowledged one-by-one.

D.4.3 Global behaviour semantics

As said earlier, the global behaviour semantics of Estelle specification is operational. This means that a, so called, *next-state relation* is defined over the set of the system *global states* which here are called *global situations*. The next-state relation (or rather *next-situation relation*) specifies all possible situations that may be directly achieved from a given situation. The overall behaviour of a system (a system defined by an Estelle specification) is then characterised by the set of all sequences of global situations which can be generated (by the next-situation relation) from a certain *initial* situation.

Recall that an Estelle specification describes a collection of systems, that modules within each system may execute their transitions in a synchronous or non-deterministic way, and that modules belonging to different systems may execute their transitions in a completely asynchronous fashion.

The systems execute in a succession of computation steps. Each computation step of a system begins by nondeterministic selection of one (in the case of a **systemactivity** system) or several (in the case of a **systemprocess** system) transitions among those *ready-to-fire* and *offered* by the system component modules (at most one transition per module may be offered at a given moment). The selected transitions are then executed in parallel. A computation step ends when all of them are completed.

From the point of view of the semantic model, the parallel execution of transitions within one computation step of a system cannot be considered simultaneous since the result may depend on whether one or another completed first (recall that, for example, these transitions may send interactions into a common queue and that the order in which they are put into this queue depends on their execution speed). All possible interleavings (permutations) of transitions selected in a computation step must therefore be taken into account in the model. Nevertheless, the execution of these transitions is synchronised in that a selection of new transitions to execute starts only when all of them have completed.

That way the relative speed of modules within a **systemprocess** system may be controlled (*synchronised*). That is why we say that the parallelism within such system has a synchronous character.

Note, however, that since only one transition is nondeterministically selected for every computation step of a **systemactivity** system, we have purely nondeterministic behaviour within such a system. In any case, the behaviour of modules with respect to each other within a system is under the control of the *system* module.

In contrast, systems run asynchronously in that their computation steps are completely independent from each other. The relative speeds of systems are not constrained (synchronised) at all.

How the transitions are selected for synchronous or nondeterministic execution within one computation step of a system depends always on the parent/children priority principle and on the way the system's modules are attributed (see D.4.3.2).

To properly model the possible behaviours of an Estelle specification, both asynchronous behaviour among systems, and synchronous and nondeterministic behaviour within a system have to be expressed by the way the transition executions are interleaved. The adequacy of this interleaved model is assured in turn by the assumption of the *atomicity* of transitions.

D.4.3.1 Global situations

Each ***global situation*** of the transition system is composed of current information on

- the hierarchical structure of modules within the specified system SP, the structure of bindings established between their interaction points, and the local state of each module. All this information is included in a *global instantaneous description* of SP (in short, gid(SP)).
- the transitions that are preselected *(currently executing)* within each system; the set of these transitions for the i-th system is denoted by Ai (i=1,..,n, where n is the total number of systems).

Each global situation is denoted by:

$$sit = (gid(SP); A1,...,An)$$

The global situation is said to be *initial* if the "gid(SP)" is initial and all sets Ai are empty. The "gid(SP)" is initial if it results from the initialisation part of the specification SP.

If Ai is empty (Ai = 0) in a global situation, then we say that the i-th system is in its *management phase*. During this phase a new set of transitions for parallel synchronous execution is selected. Otherwise, i.e., if $Ai \neq 0$, the i-th system is executing.

D.4.3.2 Next-situation relation

This relation defines the successive situations of an arbitrary current situation

$$(gid(SP);\ A1,..,Ai,..,An).$$

It is defined in the following manner:

For every i = 1,2,..,n,

1) If, in the current situation, Ai=0, then the following is a next situation

$$(gid(SP);\ A1,..,AS(gid(SP)/i),..,An)$$

where AS(gid(SP)/i) is the set of transitions selected for execution by the i-th system,

2) If, in the current situation, Ai ≠ 0, then for each transition t of Ai, the following is a next situation
(t(gid(SP)); A1,..,Ai-{t},..,An)

i.e., the new gid(SP) results from execution of transition t and t is removed from the set Ai.

Each transformation of a given global situation into a successive situation expresses the result of either a spontaneous evolution (case (1)) or an execution (or rather termination of the execution) of a transition selected among those currently executing (case (2)). By spontaneous evolution a system, which terminated all transitions previously selected, selects new transitions among those offered. Conceptually the selected transitions are considered as executing. As any transition of Ai (for any i) may terminate before any other (the relative speed of execution of transitions is not known), all of the successive situations (for each t of Ai and for each i) have to be considered. These transformations applied to the initial global situation define all possible sequences of global situations.

The execution of a transition "t" of a module:

- may cause a change in the module's local state. In particular it may modify a local variable or a control state, it may create a new child module and/or a new communication link (by executing an **attach** or **connect** statement). The transition may also generate an output (by executing an **output** statement), i.e., it may send an interaction that is appended to the FIFO queue of another module, etc. All these changes are expressed by t(gid(SP)).
- cannot influence either the choice of transitions already preselected by the other systems (the sets Aj, for j≠i, remain unchanged in the next situation) or the choice of transitions within the same system (the set Ai becomes Ai-{t} in the next situation).

The selection of transitions to be executed within one computation step, by an i-th system, i.e., the choice of the set AS(gid(SP)/i) is regulated by

- the principle of parent/children priority, and
- the modules' attributes.

The *parent/children priority principle*, which extends to the ancestor/descendant priority principle by transitivity, means that a ready-to-fire transition of a module prohibits the selection of transitions of all its descendants' modules.

The transition selection rule applied to a module within a system can be formulated as follows:

- if the module has a ready-to-fire (fireable) transition to offer, then this one will be selected (parent/children priority),
- otherwise, depending on whether the module is attributed **process** (**systemprocess**) or **activity** (**systemactivity**), respectively, all or one (chosen nondeterministically) of those ready-to-fire transitions offered by its children modules, will be preselected.

This rule applied recursively, starting with the root (system) module of the i-th system, gives the selected set AS(gid(SP)/i).

It is worth noting that the ready-to-fire transition offered by a module cannot be selected if any of this module's ancestors had something to offer. This property excludes parallelism between modules in an ancestor/descendant relation.

The application of the above rule for different systems (differently attributed modules) is illustrated in figure A.15.