ISO STANDARDIZED DESCRIPTION TECHNIQUE E S T E L L E

S. BUDKOWSKI*, P. DEMBINSKI*, M.DIAZ**

***BULL S.A.**, CORPORATE NETWORKING AND COMMUNICATION (DRCG)
DISTRIBUTED SYSTEM ARCHITECTURE AND STANDARDS (ARS)
68, route de Versailles, F-78430 LOUVECIENNES, FRANCE
phone:(33.1) 39 02 45 59

****LAAS/CNRS**, 7, Av. du Colonel-Roche, F-31077 TOULOUSE CEDEX
phone: (33) 61 33 62 56

ABSTRACT

Estelle is a Formal Description Technique, defined within ISO (International Organization for Standardization) for specification of distributed, concurrent processing systems. In particular, Estelle can be used to describe the services and protocols of the layers of Open Systems Interconnection (OSI) architecture defined by ISO. Its present ISO status is Draft International Standard (DIS 9074). The International Standard status is expected before the end of 1988. The paper (based on [36] and [37]) outlines basic concepts as well as syntactic and semantic aspects of this description technique.

Keywords: Estelle, Formal Description Technique, FDT, Distributed Systems, Concurent Systems, Specification Language, Comunication Protocols and Services, Open System Intrconnection, OSI

RESUME

Estelle est une technique de description formelle définie à l'ISO (International Organization for Standardization) pour la spécification de systèmes concurrents distribués. En particulier Estelle est bien adapté pour la description des services et des protocoles tels qu'ils sont définis dans l'architecture en couche du modèle OSI d'interconnexion des systèmes ouverts de l'ISO. Son statut actuel à l'ISO est celui de standard international provisoire (Draft International Standard - DIS 9074). Le statut définitif de Standard International est prévu pour la fin de l'année 1988. Ce papier (fondé sur les publications [36] et [37]) retrace les concepts de base ainsi que les aspects syntaxiques et sémantiques de cette technique de description.

Mots-clefs: Estelle, Technique de description formelle, FDT, Systèmes distribués, Systèmes concurrents, Langage de specification, Protocoles et services de communication, Interconnexion des systemes ouverts, OSI

0. 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.

The development of Estelle was initiated within the International Organization for Standardization (ISO) in 1981 (ISO/TC97/SC21/WG1/FDT subgroup B chaired by Richard TENNEY). It is presently ISO Draft International Standard (DIS) (the International Standard status is expected before the end of 1988). Its formal syntax and semantics are described in ISO 9074 [18] which is available from the ANSI Secretariat and from national standard organizations participating in ISO.

Estelle is a second generation of formal description techniques as it reflects the experience gained from experiments in using predecessor description techniques (see [1], [2], [3], [4] and [31]).Estelle also reflects collaboration with CCITT which defined SDL (Specification and Descriptions Language - [5]) with which Estelle has some notions in common.

Estelle can be briefly described as a technique based on an extended state transition model i.e., a model of a nondeterministic automaton extended by means of Pascal language. More precisely, Estelle may be viewed as a set of extentions to ISO Pascal [29], level 0, which models a specified system as a hierarchical structure of automata which :

- may run in parallel, and
- may communicate by exchanging messages and/or by sharing (in a restricted way) some variables,

Estelle permits to separate the description of the communication interfaces between components of a specified system from the description of the internal behavior of each such component.

This paper (based on [36] and [37]) presents the Estelle language by describing its principal concepts (section 1), the syntactic (section 2) and the semantic aspects (section 3), and by illustrating its use through examples (section 4).

1. A BRIEF OVERVIEW OF ESTELLE PRINCIPAL CONCEPTS

1.1. Modules and module instances

A distributed system specified in Estelle is composed of several communicating components. Each component is specified in Estelle by a **module definition**. Since in a system there may be more than one component defined (textually) by the same module definition, it is appropriate to call system's components **module instances**. Further on, the term **module** will be used rather than module

instance unless it can lead to a confusion.

The internal structure and behavior of a module are either defined explicitely or left for further refinement. The module definition consists of the set of actions (transitions) of a state transition system (see Section 2.2.3) the module may perform and/or the definitions of its submodules (children modules - see Section 1.2) together with their interconnections (see Section 1.3).

A module is **active** if its definition includes at least one transition; otherwise, it is **inactive**. A module may have one of the following **class attributes**

- systemprocess or process
- systemactivity or activity

or may be not attributed at all. The modules attributed "systemprocess" or "systemactivity" are called **system** modules.

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

- interaction points
- channels
- interactions

Each module has a number of input/output access points called **interaction points**. There are two categories of interaction points: **external** and **internal**. To each interaction point a channel is associated which defines two sets of interactions. These two sets consist of interactions which can go out or come in, respectively, through this interaction point. Interactions are abstract events (messages) exchanged with the module environment (through external interaction points) and with children modules (through internal interaction points).

A module is represented graphically as a box (rectangle) with points on its boundary (external interaction points) and/or inside of it (internal interaction points). The module name and its class, the name of interaction points and their associated interactions (going in or out) may be added as in Fig.1.

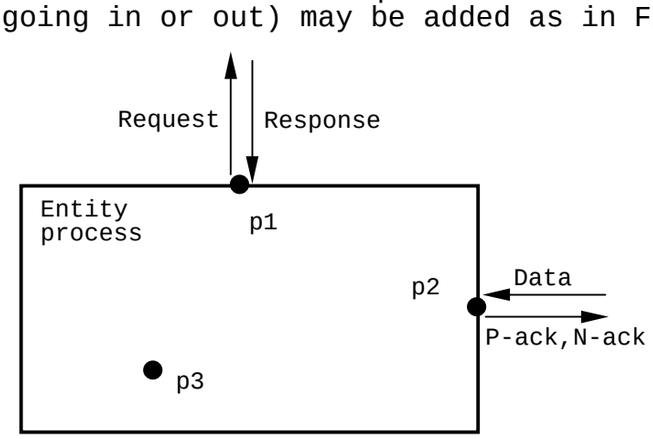

Fig. 1

1.2. Structuring

A module definition in Estelle may include definitions of other modules. This applied repeatedly, leads to a hierarchical tree structure of module definitions. Estelle provides means to create instances of child modules defined within the module definition. Note that a number of instances of the same module may change **dynamically** since thay may be created and destroyed.

The hierarchical tree structure of modules (or module instances) may be depicted as in (Fig.2a) or as in (Fig.2b). The modules are represented by boxes. The parent/children relationship is represented by edges or nested boxes, respectively. The root of the tree (or the largest enclosing box) is the main module representing the specified system. It is assumed that one (and only one) instance of main module always exists.

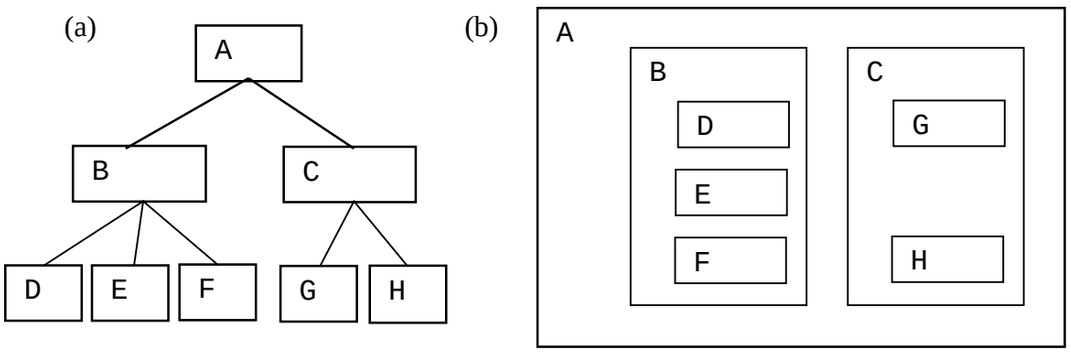

Fig. 2

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

(1) Every active module must be attributed,
(2) System modules cannot be nested within an attributed module,
(3) Modules attributed "process" or "activity" must be nested within a system module,
(4) Modules attributed "process" or "systemprocess" may be substructured only into modules attributed either "process" or "activity",
(5) Modules attributed "activity" or "systemactivity" may be substructured only into modules attributed "activity".

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

The attributing principles play an important role in defining the behavior of a specified system (see Sections 3.2).

Within a specified system, a fixed number of **subsystems** is distinguished. Each subsystem is a subtree of modules instances rooted in a system module instance (i.e., attributed "systemprocess" or "systemactivity"). In particular, the whole specified system may form just one subsystem. In such a case, the specification itself (main module) has the attribute "systemprocess" or "systemactivity".

1.3. Communication

Module instances within the hierarchy can communicate. Two communication mechanisms can be used in Estelle:

- message exchange,
- restricted sharing of variables.

1.3.1. Message exchange

The module instances may exchange messages, called **interactions**. A module instance can send interactions 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 exclusively belongs to the single interaction point (so called **individual queue**) or it is shared with some other interaction points of a module (so called **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 a so-called **communication links** between modules' interaction points, are specified. A communication link between two interaction points is composed of exactly one "connect" segment and zero or more "attach" segments. Each link segment ("connect" or "attach") will be represented graphically by line segments which bind modules' interaction points. Fig.3 illustrates this convention.

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 Fig.3 pairs of interaction points: (1,3),(2,8),(9,11),(10,13) and (17,18) are attached.

Two bound interaction points are said to be **connected** if both are external interaction points of two sibling modules (e.g., (5,6),(4,7),(1,9) and (2,10) in Fig.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 Fig.3), or both are internal or external interaction points of the same module (e.g., (15,16) and (18,19) in Fig.3).

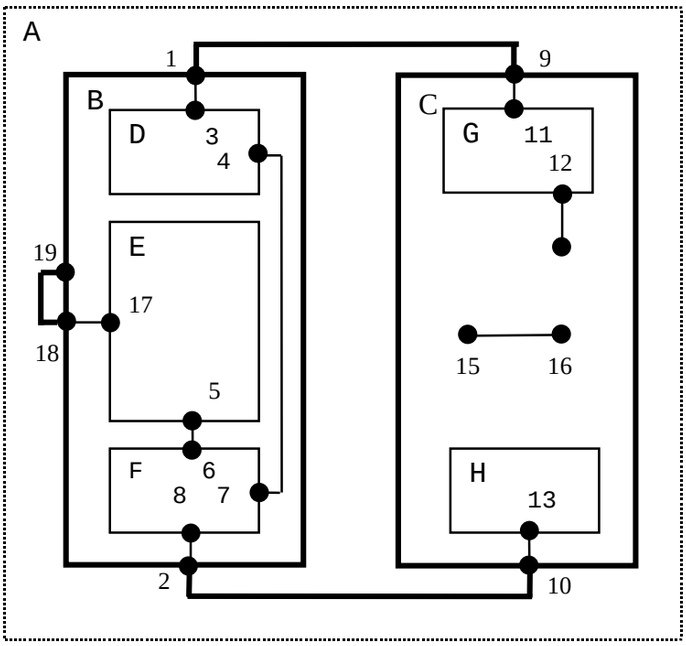

Fig. 3

It has to be noted that, at a given moment,

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

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

(3) 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 modules,

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

The communication link specifies that two modules instances, whose interaction points are the end-points of the link, can communicate by exchanging messages (in both directions) through these linked interaction points. In Fig.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 points 1,9,2 and 10 are not end-points.

An interaction sent through an interaction point that is end-

point of a communication link, directly arrives to the other end-point of this link and is always accepted by the receiving module, since the associated queue is unbounded. An interaction sent through an interaction point that is not end-point of a communication link, is considered to be lost.

1.3.2 Restricted sharing of variables

Certain variables can be shared between a module and its parent module. These variables have to 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 have always priority (so called **parent/children priority principle** of Estelle - see Sections 3.2).

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

1.4 Dynamism

Since modules enclosing subsystems are always inactive, the structure of the subsystems and their communication links, once initialized, cannot be changed (i.e.,it is **static**).

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

- system modules (subsystem roots) and their communication links are in bold lines (static system architecture),
- dotted lines are used for modules enclosing subsystems,
- nonbold lines are reserved for remaining modules and links.

The internal structure of each subsystem and bindings between interaction points of their submodules may vary (i.e., it is **dynamic**). This is because actions of a module instance within a subsystem may include statements creating and destroying its children and/or bindings between children interaction points or bindings between the module instance and its child interaction points.

1.5 Parallelism and non-determinism

Two kinds of parallelism between modules can be expressed in Estelle:

- asynchronous parallelism
- synchronous parallelism

Asynchronous parallelism is permitted only between subsystems, or more precisely, between actions of different modules' instances belonging to different subsystems.

The synchronous parallelism is permitted only within a subsystem, or more precisely, between actions of different modules' instances belonging to the same subsystem.

1.6 Typing

All manipulated objects are strongly typed. Pascal typing system is extended to purelly Estelle objects such as: modules variables, interactions, interaction points and (control)state**s.**

1.7. Module internal behevior representation

The internal dynamic behavior of an Estelle module is characterized in terms of a nondeterministic state transition system, i.e., by defining the set of states, the initial state and the next-state relation. A 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 a status of the module internal structure (submodule instances, bindings between interaction points, etc.). An initial state of a module instance is defined by an initialization 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 state described above and may output interactions to the module environment. To define transitions' actions the Pascal compound statements are used.

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

The well known model of finite state automaton (FSA) is a particuliar case of a state transition system. Hence, a FSA may be described in Estelle (see Section 4.1).

1.8 Global behavior representation

To describe the behavior of a complete system specified in Estelle, the operational style (operational semantics) has been used. This global semantics are described in more detail in Sec.3 (see also [8],[10] and [18]).

2. HIGHLIGHTS ON THE SYNTAX OF ESTELLE

2.1. Channels and interaction points

Channels in Estelle are "abstract" objects whose definitions specify sets of interactions (messages). The interaction points, through which interactions go out and come in, refer (in their declarations) 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(ROLE1, ROLE2);

by ROLE1:
   m1; m2; ... ; mN;

by ROLE2:
   n1; n2; ... ; nK;
```

where m1,..,mN, n1,..,nK 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 of the above form. In absence of parameters the interaction-identifier represents itself.

Now, an interaction point p1 may be declared as follows

```
p1 : CHANNEL_ID(ROLE1)
```

and another interaction point p2,

```
p2 : CHANNEL_ID(ROLE2)
```

In the first case, the set of interactions which can be sent via p1 contains all interactions specified for ROLE1 in the channel definition (i.e., the interactions declared by m1,m2,..,mN), and the set of interactions which can be received contains all interactions specified for ROLE2 (i.e., the interactions declared by n1,n2,..,nK).

In the second case we have, as it is easy to guess, an exact

opposite assignment of sent and received interactions, 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 (connected) must 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 second.

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

```
        p1 : CHANNEL_ID(ROLE1) common queue
or
        p1 : CHANNEL_ID(ROLE1) individual queue
```

2.2. **Modules**

A **module** is specified in Estelle by a pair which consist of

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

A **module header** definition specifies the **module type** whose values are 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 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 of the module header type is created (initialized). The following is an example of a module header definition:

```
module  A  systemprocess (n : integer);
 ip  p : T(S) individual queue;
    p1 : U(S) common queue;
    p2 : W(K) common queue;
 export   X,Y : integer; Z : boolean
end;
```

Observe that by the above definition the same 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.

At least one **module body** definition is declared for each module header definition. 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" **end**;

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 (systemprocess). But their behaviors, defined by the body definitions may be different. This means that modules may have different behaviors and the same external visibility. A body defined as "external" does not denote any specific behavior of the module. It indicates that either the module body definition already exists elsewhere or will be provided later in the process of specification refinement. The "external" bodies nicely serve to allow describing an overall system architecture without detailed description of the system components. This feature is illustrated in Section 4.2.

The body definition is composed of three parts:

- declaration part
- initialization part
- transition part

2.2.1. Declaration part

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

- channels
- modules
- module variables
- state and state-sets
- internal interaction points

Note that 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 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 section. The hierarchy of the module definitions is depicted in

Fig.4.

```
module  A...end;
body B for A;

   module  A1  process;
   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;
```

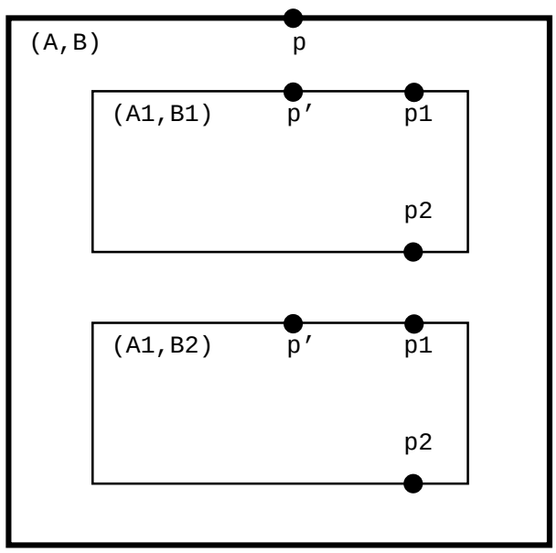

Fig. 4

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

modvar X,Y,Z : A1

may occur which 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** and **release** statements, see Sections 2.2.2 and 2.2.3).

The internal behavior of each module (instance) is defined in terms of a state transition system (see Sec. 1.7). 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.

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

stateset IDWA = (IDLE, WAIT)

The **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 within a module header.

2.2.2. Initialization part

The initialization part of a module body, indicated by the keyword "initialize", 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 initialized which means that the module's children can be created. Creation of children module instances during initialization defines their "initial architecture"

To initialize Pascal variables, Pascal statements are used (for example, X := 5) and to initialize the "state" variable to a control state, for example IDLE, we write **to** IDLE.

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

```
initialize
 begin
  init  X  with  B1;
  init  Y  with  B2;
  init  Z  with  B1;
  connect X.p1 to Y.p2;
  connect Y.p1 to Z.p2;
  attach p to X.p';
 end;
```

The above initialization 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 as being of 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 concrete hierarchy of module instances of Fig.5 corresponds to the hierarchical "pattern" of module definitions from Fig.4.

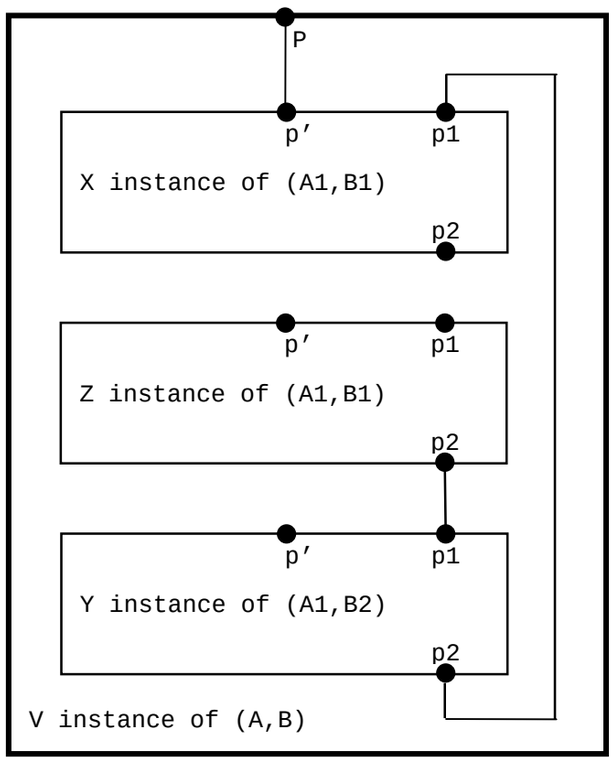

Fig. 5

The initialization also establishes bindings between appropriate interaction points of the three newly created module instances . These bindings are also presented in Fig.5. Recall that two interaction points must play opposite roles in order to be connected (because the exchange of interactions is made via these points), and must play the same role in order to be attached (because the interaction point of a child replaces that of the parent).

2.2.3. Transition part

The transition part describes, in detail, the internal behaviors of modules. It is composed of a collection of transitions declarations. Each transition begins with the keyword "trans". A transition may be either **simple** or **nested**. A nested transition is a short hand notation for a collection of simple transitions (see below). The following is an example of a nested transition:

```
trans when N.NI(p)
       from IDLE to WAIT
          provided p.present
            begin
             keep_copy(p,saved);
             output U.UC
            end;
          provided otherwise
            begin
             output N.NR
            end;
       from IDLE to SAME
        begin
        end;
```

Below we give the expansion of the above nested transition into simple transitions:

```
trans when N.NI(p)
       from IDLE to WAIT
        provided p.present
         begin
          keep_copy(p,saved);
          output U.UC
         end;
trans when N.NI(p)
       from IDLE to WAIT
        provided not(p.present)
         begin
          output N.NR
         end;
trans when N.NI(p)
       from IDLE to SAME
        begin
        end;
```

One can see that in the above expanded form there is exactly one "begin-end" block associated with each keyword "trans". This characterizes simple transitions. Algorithms to verify that nested transitions are well-formed so they may be expanded properly, say by a compiler, are proposed and analyzed in [24]. Similar algorithms are parts of existing Estelle compilers.

Each **simple** transition declaration is composed of two parts :

- the transition condition;
- the transition action.

The **transition condition** is composed of one or more clauses of the following categories:

- **from**-clause (**from** A1,..,An, where Ai is a control state or control state-set identifier);
- **when**-clause (**when** p.m, where p is an interaction point and m an interaction);
- **provided**-clause (**provided** B, where B is a boolean

expression);
- **priority**-clause (**priority** n, where n is a non-negative constant);
- **delay**-clause (**delay**(E1,E2), where E1 and E2 are non-negative integer expressions).

Some clauses may be omitted and at most one of each category may appear in the condition of a simple transition. 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 **delay transition**.

A **from**-clause is said to be **satisfied** in a module state if the current value of the module's control variable "state" is among those listed by the from-clause. For example, if IDLE is the current control state of a module, then all three of the following from- clauses are satisfied:

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

(recall that IDWA=(IDLE,WAIT)).

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 transition condition, are 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 delay 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".

In summary, the condition of a transition decides whether the transition is **firable** (or **ready-to-fire**) in a module state (and at a given moment of time if it concerns a delay transition). The action of one of those firable transitions eventually executes and the module will reach a new state.

The transition action is composed of two parts:

- a **to**-clause (**to** A, where A is a control state identifier),

- a transition block, i.e., a sequence of Pascal statements (with specific Estelle extensions and restrictions) between "**begin**" and "**end**" keywords.

The "**to**-clause" (e.g. **to** OPEN) specifies the next control state (OPEN) which will be attained once the transition is fired. If omitted the next state is the same as the current state.

The Pascal extensions consist of additional statements which make it possible to create and destroy module instances, to create and destroy bindings between interaction points, and to send interactions.

The "create" statements are those described previously (**init**, **connect**, **attach**). The statements for the "destroy" counterparts are:

release X;
disconnect p;
disconnect X;
detach p;

In the "**disconnect**" and "**detach**" statements (similarly in "**attach**" and "**connect**" statements) p may refer to either an interaction point of the module issuing the statement (i.e., p is an interaction-point-identifier) or to an interaction point of one of its children modules. In this second case the interaction point is accessed by the form "X.p1" ("**detach** X.p1" or "**disconnect** X.p1") which means that the statement concerns the interaction point named p1 of the module currently referenced by the module variable X.

The "**release** X" statement destroys the module referenced by the module variable X and all its descendant modules.

The "**disconnect** p" statement disconnects the interaction point p from the interaction point to which it was connected , and "**disconnect** X" disconnects all the interaction points of the children module referenced by X.

The "**detach** p" statement detaches the interaction point p from the interaction point it was attached to.

If two interaction points p1 and p2 are connected (attached), then the result of "**disconnect** p1" ("**detach** p1") is the same as that of "**disconnect** p2" ("**detach** p2").

The ability to execute these statements within a transition gives the possibility to change dynamically the hierarchical tree structure of modules as well as the communication links between them.

There is also a special statement **output** which allows a module to send an interaction via a specified interaction point. For example, the statement "**output** p1.m" sends the interaction m via the interaction point p1.

As we explained earlier (see Sec.2.3), if p1 and p2 are the two end-points of a communication link, then the "**output** p1.m" statement leads to appending interaction m in the queue associated with the interaction point p2.

The **restrictions** to Pascal [29] adopted in Estelle are mainly the following:

- all declared functions are "pure" i.e., without side effects,
- pointers may be used only in purely Pascal constructs,
- conformant arrays cannot be used,
- file type cannot be used,
- goto statements and labels are restricted in use,
- read and write statements cannot be used.

2.3. **Specification module**

All modules defined as described in the preceding sections are textually embodied in a principal 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 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 specification for which this assignment is omitted in their declarations.

The time-option defines the unit of time (milisecond, 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.

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 which communicates with other modules. In practice, a specification body often constitutes a general "framework" for an open system being defined, i.e., it provides a global context necessary for the system definition and initialization.

3. AN OVERVIEW OF THE ESTELLE SEMANTICS

As said earlier, the semantics of Estelle 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 behavior of a system (a system defined by an Estelle specification) is then characterized by the set of all sequences of global situations which can be generated (by the next-situation-relation) from a certain **initial** situation.

3.1. **Global situations**

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

- the hierarchical structure of module instances within the specified system SP, the structure of bindings established between their interaction points, and the local state of each module instance. All this information is included in a, so called, **global instantaneous description** of SP (in short gid(SP)).

- the transitions that are in "parallel (synchronous) execution" within each subsystem; the set of these transitions for i-th subsystem is denoted by Ai (i=1,..,n, where n is the number of subsystems).

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 initialization part of the specification SP.

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

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 is empty (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 subsystem,

2) If, in the current situation, Ai is non empty (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 the 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 completing the execution) of a transition selected among those currently executing (case (2)). 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 (computations).

How the set of transitions (i.e., AS(gid(SP)/i)) is selected for synchronous or non-deterministic execution within one computation step of an i-th subsystem, depends always on the parent/children priority principle and on the way the subsystem's modules are attributed (see Sec.1.2).

The **parent/children priority principle**, which extends to the ancestor/descendent priority principle by transitivity, means that a ready-to-fire action of a module prohibits the selection of actions of all its descendent modules.

The role of attributes is best illustrated by two particular cases. The first when all subsystem's modules are attributed

"process" (the system module is attributed "systemprocess") and the second when all of them are attributed "activity" (the system module is attributed "systemactivity").

In the first case **all** (but at most one per module) ready-to-fire transitions (actions) that are not in the ancestor/descendent conflict, are selected (Fig.6a), while in the second case **only one** of them (Fig.6b) is selected. Therefore, in fact, there is no synchronous parallelism within a computation step of a "systemactivity" subsystem. The subsystems behaves in a non-deterministic manner. The intermediate selections, between the above two extremes, are possible due to the fact that a "process" ("systemprocess") module may be substructured in both "processes" and "activities" (Fig.6c).

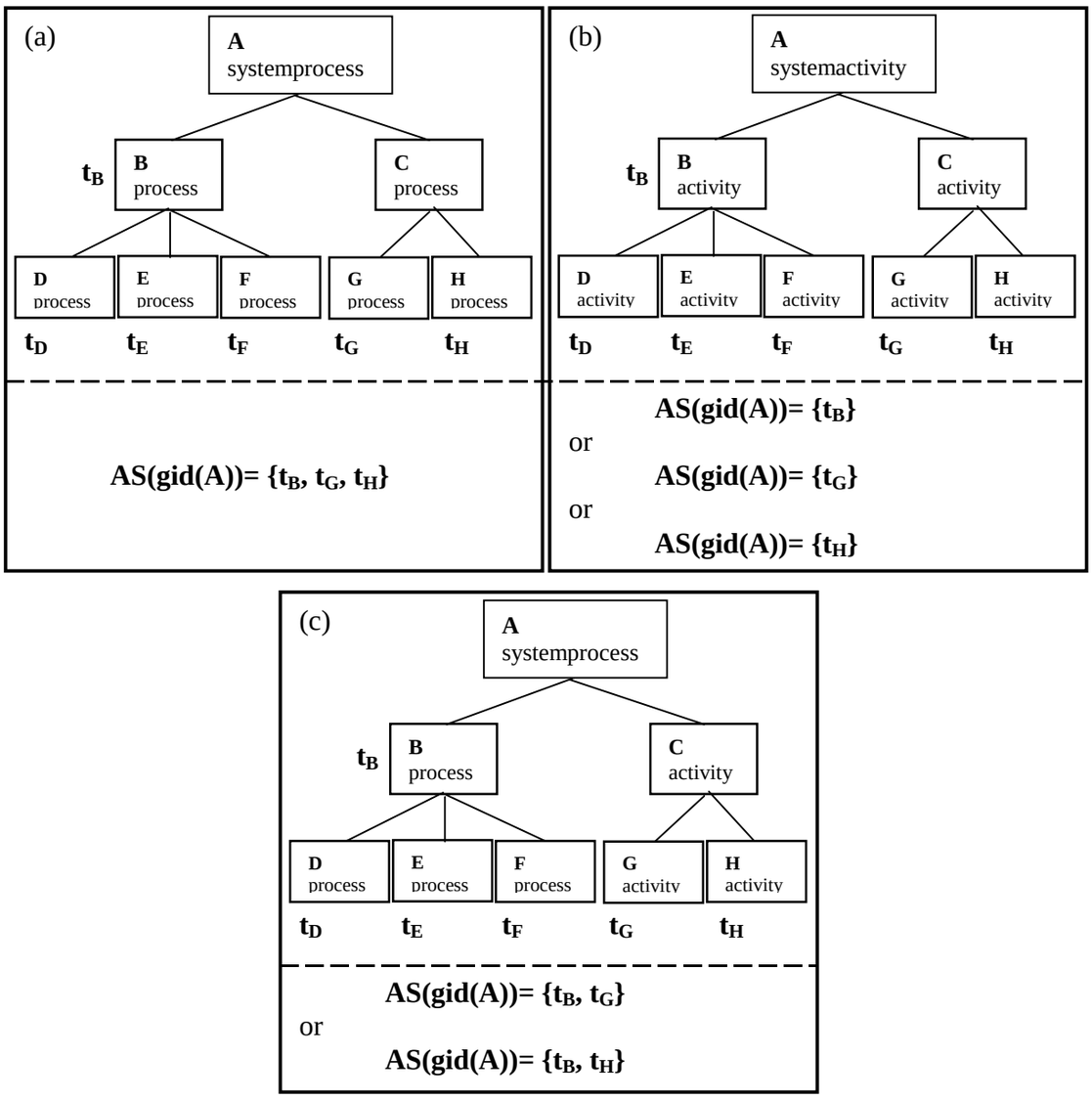

Fig. 6

4. EXAMPLES

The examples below are not specifications of real systems. They help the reader become familiar with the syntax of Estelle and:

- illustrate how an Estelle module represents a Finite State Automaton (FSA) and how an introduction of variables and parameters may shorten the description (Sec.4.1),

- illustrate a way the overall structure of a system can be specified using "external" parameters which replace explicit module body definitions (Sec. 4.2),

4.1 Estelle representation of a finite state automaton (FSA)

In the examples below we show how to describe a behavior of a module in terms of a simple state automaton (specification Example1) and how to express an equivalent behavior (but in a more concise way) when we allow some extention to the FSA model (specification Example2).

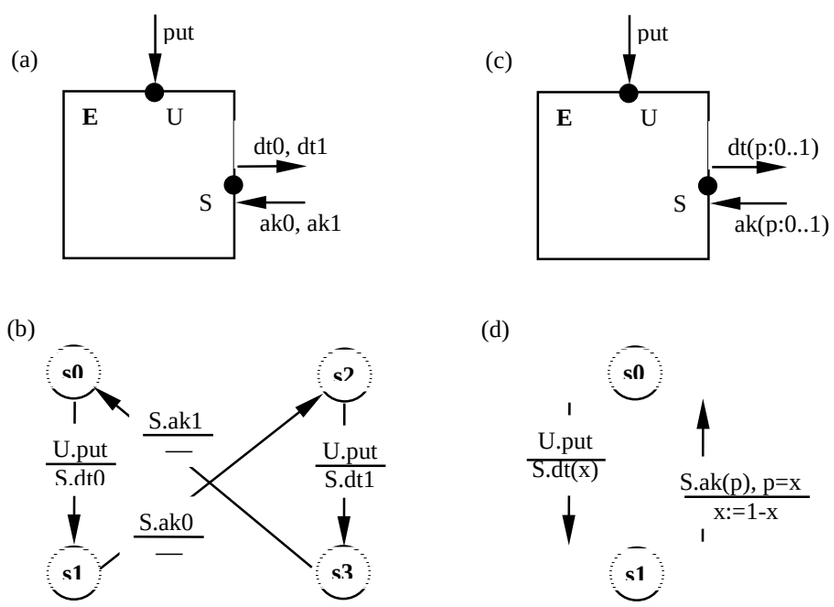

Fig. 7a, 7b, 7c, 7d

The specified system is composed of just one module E. In Fig.7a we show the module and its interface with the environment, while Fig.7b depicts the FSA graph representing the module intended behavior. The Estelle specification is the following:

```
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;
  trans  when S.ak0
          from s1  to s2
            begin  end;
  trans  when S.ak1
          from s3  to s0
            begin  end;
end;
end.
```

An equivalent behavior may be represented by the following Estelle description:

```
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;
  trans  when S.ak(p)
          provided  p=x
           from S1  to S0
             begin  x := 1-x  end;
end;
end.
```

Note that in the above description two extentions have been made. The first is the parameter "p" of the enumerated type T = 0..1

which permits to declare two-element classes "dt" and "ak" of interactions insted of declaring their elements as separate four interactions (observe that "dt(0)" corresponds to "dt0" in Example1, etc.). The second extention is the variable "x" of the same type T. It permits, in a similar way, to replace previous four control states by only two (observe that in Example2, the situation of being in the control state "S0" with x=0 corresponds to to the situation of being in the control state "s0" in Example1, etc.).

The graphical representation of the module's interface with the environment and of the module's internal bebavior in Example2, is shown in Fig.7c and Fig.7d, respectively.

4.2 **Specifying the overall structure of a system**

The specification below declares and initializes a system which consists of three subsystems X,Y and Z (i.e., the subsystems are referenced by module variables X,Y and Z of types: USER,RECEIVER and NETWORK, respectively). These subsystems exchange some messages through their interaction points connected as declared in the initialization part of the specification. The scheme of the specified EXAMPLE system is presented in Fig.8.

```
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;
 module NETWORK systemprocess;
    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.
```

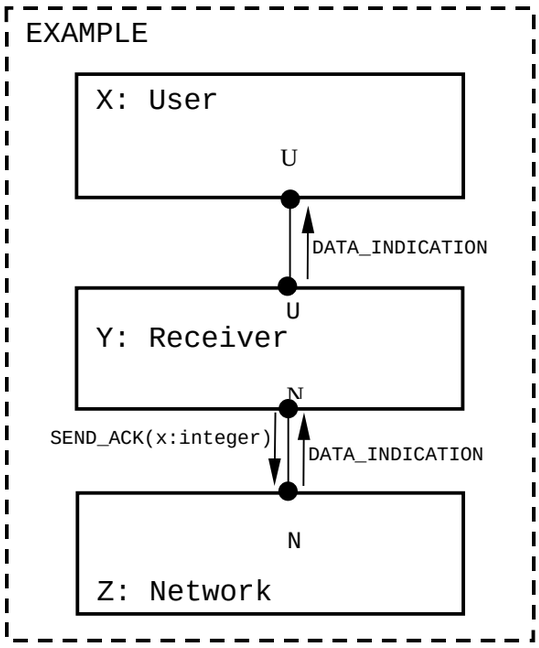

Fig. 8

5. CONCLUSION

An overview of the most important concepts and facilities of Estelle has been presented (for more information see [7],[15],[19],[20],[36],[37] or consult [18]).

The readers interested in real-life application examples may examine several ISO protocols and services which have been described in Estelle (see [23], [25], [26], [27], [32], [35],[43], and overview papers [19] and [28]). It has to be noted that some of these descriptions make use of earlier versions of Estelle. Nevertheless, they constitute quite rich material for further experiments.

It is appropriate to end this presentation emphasizing the importance of the Estelle supporting tools. Given the size of existing Estelle specifications, it seems clear that without such tools real-life specifications cannot efficiently be developed and reliably implemented. Some of these tools already exist and are tested. Other are being under development. Their descriptions are obviously outside of the scope of this paper. Some information is available in the open literature and in technical reports (e.g., [7],[9],[11]-[17],[21],[22],[28],[30],[33],[34],[38] and [40]-[42]). Within the ESPRIT european SEDOS/ESTELLE/DEMONSTRATOR project an ESTELLE WORKSTATION integrating all basic Estelle tools (editor, compiler and simulator) is currently under development and its evaluation on many real-life applications (in the domains of Computer Networks, Telecomunication, Space

Protocols and Industrial Control Systems) is also conducted [35].

REFERENCES

[1] ANSART J.P., RAFIQ O., CHARI V., Protocol Description and Implementation Language (PDIL), Proc. IFIP 2nd Workshop on Protocol Specification, Verification and Testing, C. SUNSHINE (ed), North Holland,1982.

[2] AYACHE J.M., COURTIAT J.P., DIAZ M; LC/1, A Specification and Implementation Language for Protocols, Proc. IFIP 3rd Workshop on Protocol Specification, Verification and Testing, H. RUDIN and C.H. WEST (ed), North Holland, 1983.

[3] BOCHMAN G.V., Finite State Description of Communication Protocols, Computer Networks, Vol.2, p.361-378, Oct. 1978.

[4] TENNEY R.L., BLUMER T.P., A Formal Specification Technique and Implementation Method for Protocols, Computer Networks, vol.6, 1982.

[5] CCITT/SGXI Recommendation Z101 to Z104, Functional Specification and Description Language, 1985.

[6] DIAZ M., SEDOS : Un environnement logiciel pour la conception des systèmes distribués, Proc. 3ème Congrès "De nouvelles architectures pour les communications", Paris, October 1986.

[7] COURTIAT J.P., DEMBINSKI P., GROZ R., JARD C., ESTELLE: un langage pour les algorithmes distribués et les protocoles. Technique et Science Informatiques, vol. 6, N° 2, 1987.

[8] DEMBINSKI P., Estelle Semantics, (SEDOS Rep. SEDOS/ 054, June 1986), also in [39].

[9] DEMBINSKI P., BUDKOWSKI S., Simulating Estelle Specifications with Time Parameters, Proc. IFIP WG 6.1 Seventh Conference on Protocol Specification, Testing, and Verification (eds. H. Rudin and C.H. West), North-Holland 1987.

[10] COURTIAT J.P., Petri Nets Based Semantics for Estelle, (SEDOS Rep. SEDOS/109, November 1987), also in [39].

[11] ANSART J.P. et al., Software tools for Estelle, Proc. IFIP 6th International Workshop on Protocol Specification, Testing and Verification, Montreal, June 10-13, 1986.

[12] DIAZ M., VISSERS CH., ANSART J.P., SEDOS, Software Environment for the Design of Open distributed Systems, ESPRIT Week, Brussels, Sept. 1985, North Holland 1986 (CEC Ed.).

[13] User Guide for the NBS Prototype Compiler for Estelle, Rep. N° ICST/APM 87-1, NBS Institute for Computer Science and Technology, 1986.

[14] Estelle Development System, Users Manuel, Phoenix Technologies Ltd. (previously Protocol Development Corporation), 675 Mass. Ave., Cambridge, MA.,1986.

[15] DIAZ M., ESTELLE, une technique de description formelle de protocole, 9eme Journees Francophones sur l'Informatique, Liège, 20-21 Janv., 1987, DUNOD (Ed. A. Danthine), 1987.

[16] S.T. VUONG A.C. LAN, Semi-automatic Implementation of Protocols, Proc. IEEE INFOCOM 87, San Francisco 1987.

[17] T. KATO, T. HASEGARA, H. HORINCHI, Design of Translator from Estelle with ASN.1 to ADA ; KDD, Information Processing Laboratory, 2-1-23 Nakameguro, Meguro-ku, Tokyo 153, 1987.

[18] ISO/TC97/SC21/WG1/DIS9074 Estelle - A formal Description Technique Based on an Extended State Transition Model, 1987.

[19] LINN R.J., Jr. Tutorial on the Features and Facilities of Estelle, ICST report, National Bureau of Standards, Gaithersburg, MD 20899, U.S.A. August 1987

[20] BUDKOWSKI S., DEMBINSKI P., ANSART J.P., Estelle, un langage de specification des systèmes distribués, Proc. 3ème Congrès "De nouvelles architectures pour les communications", Paris 28-30 October 1986.

[21] RICHIER J.L., RODRIGUEZ C., SIFAKIS J., VOIRON J. Verification in XESAR of the Sliding Window Protocol, Proc. IFIP WG 6.1 Seventh Conference on Protocol Specification, Testing, and Verification (eds. H. Rudin and C.H. West), North-Holland 1987.

[22] JARD C., GROZ R., MONIN J.F., VEDA: a Software Simulator for the Validation of Protocols Specifications, Proc. COMNET'85, Budapest, North-Holland 1985.

[23] AMER P. D., CECELI F., JUANOLE G., Formal Specification of ISO Virtual Terminal in Estelle, CIS Dept. Tech. Report 87-12, University of Delaware, August 1987.

[24] AMER P. D., SRIVAS M., PRIDOR A., Semantic Well-Formedness of Estelle Specification Transitions, Proc. IFIP WG 6.1 Eighth Symp. on Protocol Specification, Testing, and Verification, Atlantic City, June 1988 (North-Holland 1989-to be published)

[25] ISO/TC97/SC6 N4394, Formal Description of ISO 8073 (Transport Protocole) in Estelle, Information Processing Systems, Open System Interconnection, 1986.

[26] ISO/TC97/SC6 N4393, Formal Description of Transport Service in Estelle, Information Processing Systems, 1986.

[27] ISO/TC97/SC6 N4542, Annex 4, Formal Description of Protocole for Providing the Connectionless-mode Network Service in Estelle, Information Processing Systems, 1987.

[28] DIAZ M., VISSERS Ch., BUDKOWSKI S., Estelle and LOTOS Software Environments for the Design of Open Distributed Systems, Proc. 4th Annual ESPRIT Conference , Brussels, Sept. 28-30, 1987, Noth-Holland, CEC Ed.

[29] ISO International Standard 7185, Programming Language - Pascal, ISO/TC97/SC6/WG4, 1983.

[30] User Guide for the BULL/MARBEN Estelle to C Compiler, BULL S.A., Corporate Networking and Communication (DRCG), Distributed System Architecture and Standards (ARS), 68,Route de Versailles, F-78430 LOUVECIENNES, 1988.

[31] VISSERS,C.A., TENNEY R.L., BOCHMAN G.V., Formal Description Techniques, Proc. IEEE, vol.71, 1983.

[32] MONDAIN-MONVAL, P., Estelle Description of the ISO Session Protocol, SEDOS Rep. SEDOS/106, Noember 1987, in [39].

[33] DE SAQUI-SANNES, P., COURTIAT J-P., ESTIM: An Interpreter for the Simulation of Estelle Descriptions, SEDOS Rep. SEDOS/115, November 1987, in [39].

[34] PAPAPANAGIOTIKAIS, G., AZEMA, P., CHEZALVIEL-PRADIN, B., On a Prolog Environment for Protocol Analysis, Proc. Conference on Distributed Computing Systems, Cambridge Ma., May 1986.

[35] AYACHE J.M.et al, Presentation of the SEDOS ESTELLE DEMONSTRATOR project, in [39].

[36] BUDKOWSKI S., DEMBINSKI P., An Introduction to Estelle: A Specification Language for Distributed Systems, Computer Network and ISDN Systems Journal, vol.14, Nb.1, 1988

[37] DEMBINSKI P., BUDKOWSKI S., Specification Language ESTELLE, in [39].

[38] User Guide for BULL simulator/debugger for Estelle, BULL S.A., Corporate Networking and Communication (DRCG), Distributed System Architecture and Standards (ARS), 68,Route de Versailles, F-78430 LOUVECIENNES, 1988.

[39] DIAZ M. et al (ed), The Formal Description Technique Estelle, North-Holland, 1989 (to be published)

[40] CHARI V. et al, The Estelle Translator, in [39].

[41] CHARI V. et al, An Estelle Simulator/Debugger Tool, in [39].

[42]RICHARD J.L.,CLAES T.,A Genaratorof C-code for Estelle,in [39]

[43] CHARI V., A Transport protocol, in [39].