An Introduction to Estelle: A Specification Language for Distributed Systems

S. BUDKOWSKI and P. DEMBINSKI
Bull S.A., Corporate Networking and Communication (DRCG), Distributed System Architecture and Standards (ARS), 68 route de Versailles, 78430 Louveciennes, France

Estelle is a Formal Description Technique, defined within ISO (International Organization for Standardization) for specification of distributed, concurrent information 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 article outlines syntactic and semantic aspects of this description technique.

Keywords: Estelle, Formal Description Technique, FDT, Distributed Systems, Specification Language, Communication Protocols and Services, Open System Interconnection, OSI

Stanislaw Budkowski received the M.S. degree in Electronics and Ph.D. degree in Computer Science from the Warsaw Technical University, Warsaw, Poland, in 1961 and 1968, respectively.

In 1961 he joined the faculty of the Department of Electronics and the Institute of Computer Science of the Warsaw Technical University where he was Associate Professor from 1971. He spent the academic year 1961/62 at the University of Grenoble, France, (fellowship of the French Government) and his sabbatical academic year 1975/76 at the University of Maryland, College Park, Maryland, USA.

In 1982 he joined the Agence de l'Informatique, Paris, France, and then, in 1985, the Division of Distributed System Architecture and Standards of Bull S.A./DRCG, Louveciennes, France, where he is managing a group working on methods and software tools supporting the specification, design and validation of communication protocols.

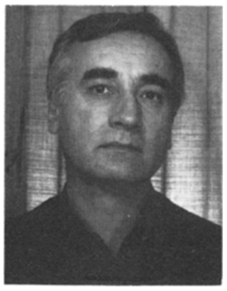
His research area included digital system design, fault-tolerant computing, microprogram specification, design and verification and communication protocols. He is an author or coauthor of several papers in these domains and coauthor of a book on digital system design.

1. Introduction

The problem of specifying distributed systems is much more difficult than that of the classical (sequential) systems. The difficulties are related to the necessity of describing several sequential components which may then cooperate and execute in parallel.

To attain good reliability of produced software, its development should begin with formal specifications. Only based on such specifications may one properly undergo verifications. It is a well known fact that the cost of error correction increases quickly during the software life-cycle. Therefore it is cost-effective to detect and correct errors during the initial phase of the software development process. This necessity is only reenforced by the design complexity when distributed systems are involved.

Estelle is a language for specifying distributed systems with a particular application in mind, namely that of communication protocols and services. The semantics for Estelle have been for-



Piotr Dembinski received the M.S. and Ph.D. degree in mathematics and computer science from the University of Warsaw, Poland, in 1966 and 1971, respectively.

From 1966 to 1971 he was an Assistant Lecturer and then Assistant Professor in the Institute of Mathematics of the Warsaw University. In 1971 he joined the Institute of Computer Science of the Polish Academy of Sciences, Warsaw. His research area included semantics of computer languages, verification of programs, microprogramming and distributed systems. He is an author of several papers in these domains and coauthor of two books on semantics of programming languages. The academic year 1974/75 Dr. Dembinski spent at the University of California, Los Angeles as a Visiting Scholar under the Senior Fulbright-Hays Program, and in 1984/85 he was a Visiting Scientist in the Department of Computer Science of the Chalmers University of Technology and University of Goteborg, Goteborg, Sweden. Since 1985 he is working within the framework of the European Project ESPRIT/SEDOS, first in the Agence de l'Informatique, Paris, then in the Division of Distributed System Architecture and Standards of Bull S.A., Louveciennes, France. His current work concerns development of methods and software tools supporting the design and validation of communication protocols.

mally defined in [8] and included in [18], justifying the claim that Estelle is a Formal Description Technique.

An Estelle specification is aimed at describing a structure of communicating state automata whose internal actions are defined by Pascal statements (with some restrictions and extensions).

Estelle is one of the description techniques which are to serve as means to remove ambiguities from ISO protocol standards, traditionally defined by a combination of a natural language prose, state tables, etc. But an unambiguous formal specification still may be far from any implementation. There is a vital need for formalized 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. We are convinced that this is precisely where the principal field of Estelle application is situated.

A considerable effort is presently being made in developing specifications in Estelle and in designing supporting tools for this language. In particular, such specifications and tools are being developed within two projects of the European Esprit program: SEDOS (Software Environment for Design Open Systems - Nov. 1984–Oct. 1987), and SEDOS-ESTELLE-DEMONSTRATOR (June 1986 - May 1989).

The prototype tools such as an (syntax driven) editor, a compiler, interpreter, and a simulator/debugger are the outcome of the SEDOS project. The Estelle Work Station will be developed within the SEDOS-ESTELLE-DEMONSTRATOR project. For further details, see [6,7,11,28,30,33] and references there.

Similar efforts in developing Estelle tools and specifications are also conducted in the USA (e.g., National Bureau of Standards [13], Protocol Development Corporation (now Phoenix Technologies Ltd.) [14], University of Delaware [23,24]), in Canada (e.g., University of Montreal [15], University of British Columbia [16]), and in Japan, e.g., [17]. At the time being at least four Estelle compiler prototypes are operational throughout the world.

Estelle has benefited from experiments in using predecessor description techniques (see [1–4,31]) and from cooperation with CCITT which defined SDL (Specification and Descriptions Language - [5]) with which Estelle has some notions in common.

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). Its present ISO status is a Draft International Standard (DIS) the last step before becoming an International Standard. 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.

This paper presents the Estelle language by describing its principal features (Section 2), the syntactic (Section 3) and the semantic aspects (Section 4), and by illustrating its use through examples (Section 5).

2. A Brief Overview of Estelle Principal Features

2.1. Tasks

A distributed system specified in Estelle is viewed as a collection of communicating components called *module instances*. For brevity, we name module instances *tasks* in this paper. Each task has a number of input/output access points called *interaction points*. Two kinds of interaction points are distinguished: *external* and *internal*. A task will be represented graphically as a rectangle with points on its boundary (external interaction points) and/or inside of it (internal interaction points). In Fig. 2 the reader can find examples of the above convention. The interaction points 1–13 of this figure are external, while points 14–16 are internal (of task C).

The internal behavior of a task is described in terms of a nondeterministic communicating state automaton (a transition system) whose transition actions are given in the form of Pascal statements (with some restrictions and extensions).

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

A task may have one of the following *class attributes*:

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

The tasks attributed “systemprocess’ or “systemactivity” are called *system* tasks.

2.2. Structuring

Tasks may be nested. The hierarchical (tree) structure of tasks may be depicted as in Fig. 1a or as in Fig. 1b. 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 task representing the specified system.

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

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

Observe that inactive tasks can be attributed, that all tasks embodying a system task are inactive and nonattributed, and that those are the only nonattributed tasks on every path going down from the root of the tree. The attributing principles play an important role in defining the behavior of a specified system (see Sections 2.4 and 4.2, and the example of Section 5.1).

Besides the hierarchical task structure, a communication structure exists within a specification. The elements of this structure will be represented graphically (Fig. 2) by line segments which bind tasks' interaction points. Only two types of binding are allowed. When an external interaction point of a task is bound to an external interaction point of its parent task, we say that these interaction points are *attached*. In Fig. 2 pairs of interaction points: (1, 3), (2, 8), (9, 11) and (10, 13) are attached. Two bound interaction points are said to be *connected* if both are external interaction points of two sibling tasks (e.g., (5, 6), (4, 7), (1, 9) and (2, 10) in Fig. 2), or one is an internal interaction point of a task and the other is an external interaction point of one of its children tasks (e.g., (12, 14) in Fig. 2), or both are internal interaction points of the same task (e.g., (15, 16) in Fig. 2).

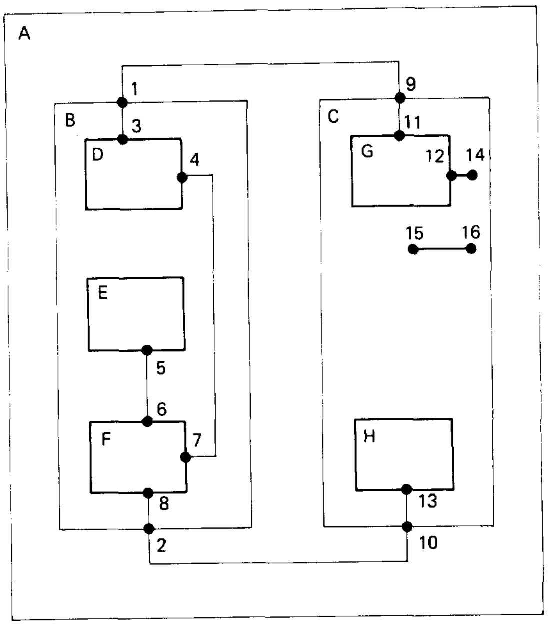


Fig. 2.

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 external point of a task may be attached to at most one external interaction point of its

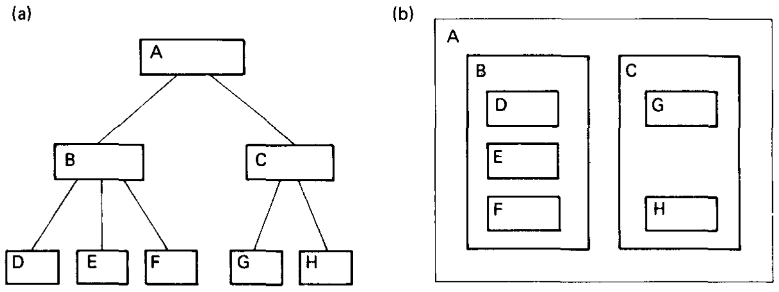


Fig. 1.

parent task and to at most one external interaction point of its children tasks,
(3) an external interaction point of a task attached to an external interaction point of its parent task cannot be simultaneously connected.

The above restrictions are illustrated in Fig. 3 (dotted and solid lines represent "connect" and "attach" bindings respectively).

A line, composed of exactly one "connect" segment and zero or more "attach" segments, forms a, so-called, *link* between two interaction points. This link specifies that two tasks, whose interaction points are the end-points of the link, can communicate by exchanging messages (in both directions) through these linked interaction points (see Section 2.3). In Fig. 2, for example interaction points 3 and 11 or 8 and 13 are end-points of links between tasks D and G, and F and H, respectively. The points 1, 9, 2 and 10 are not end-points.

Within a specified system, a fixed number of subsystems and communication links between them are distinguished. Each subsystem is a subtree of tasks rooted in a system task (i.e., a task attributed "systemprocess" or "systemactivity"). In particular, the whole specified system may form just one subsystem. In such a case, the specification itself (main task) has the attribute "systemprocess" or "systemactivity".

For clarity of presentation, the following conventions are assumed in subsequent figures:
- dotted lines are used for tasks enclosing subsystems,
- system tasks (subsystem roots) and communication links between them are in bold lines,
- nonbold lines are reserved for remaining tasks and links.

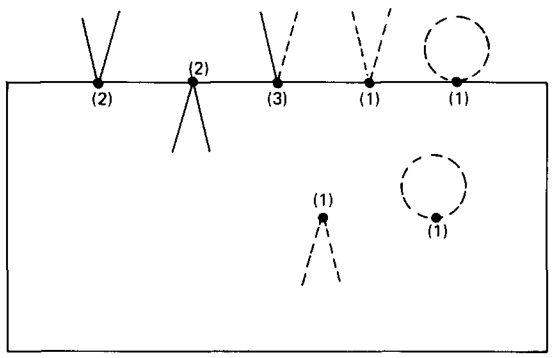


Fig. 3.

Fig. 4 illustrates these conventions.

Since tasks enclosing subsystems are always inactive, the structure of subsystems and their communication links, once initialized, cannot be changed (i.e., the structure is *static*). In contrast, the internal structure of each subsystem may vary (i.e., the structure is *dynamic*). This is because actions of a task may include statements creating and destroying its children tasks and their communication links.

2.3. Communication

Two communication mechanisms can be used in Estelle to enable cooperation between tasks:
- message exchange,
- restricted sharing of variables.

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

A task can always send an interaction. This principle is sometimes known as non-blocking send communication as opposed to blocking send also known as rendez-vous communication.

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 task, 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. Thus only "end-to-end" communication between tasks is possible.

Certain variables can be shared between a task and its parent task. These variables have to be declared as *exported* variables by the children task. This is the ONLY way variables may be shared. The simultaneous access to these variables by both a parent and child is excluded because the execution of the parent's actions have always priority (so called *parent/children priority principle* of Estelle, see Sections 2.4 and 4.2).

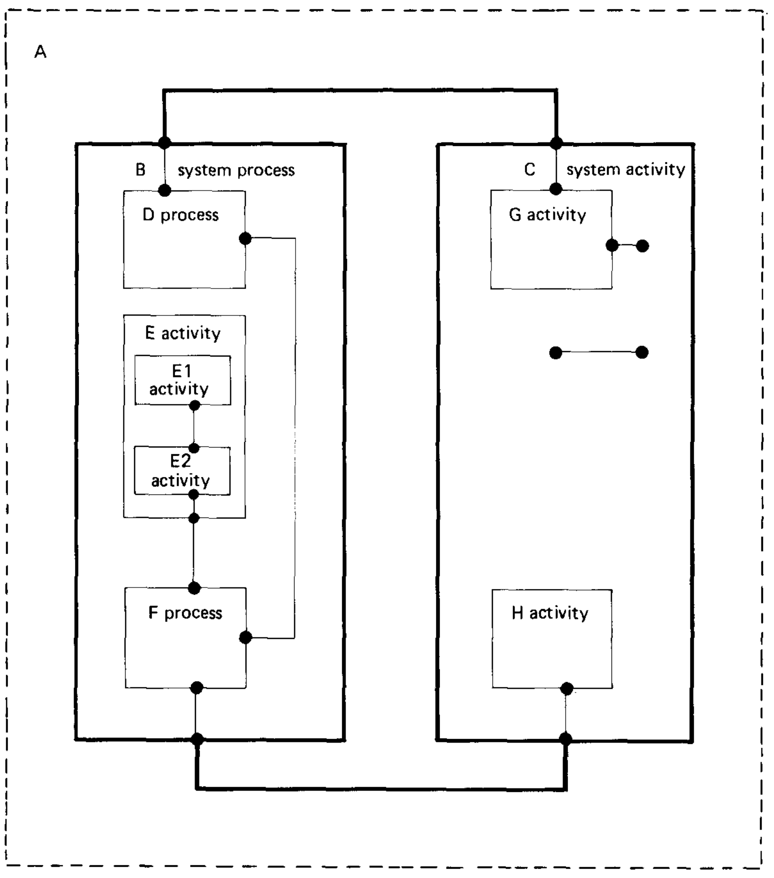


Fig. 4.

2.4. Parellelism and Non-Determinism

Two kinds of parallelism between tasks can be expressed in Estelle:
- asynchronous parellelism,
- synchronous parellelism.

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

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

Intuitively speaking, each subsystem runs by its own *computation steps*. A computation step begins by a selection of a set transitions (actions) among those *ready-to-fire*, or *offered-to-fire* by the subsystem's tasks (at most one transition per task). Then these transitions are executed (in parallel) and, when all of them have completed, the next computation step begins. That way the relative speed of tasks within a subsystem may be, in general, controlled (synchronized). That is why we say that the parallelism within a subsystem has a synchronous character. Note, however, that if the selected set consists of exactly one action for every computation step, then we have purely non-deterministic behavior within a subsystem.

How the set of transitions (actions) is selected for synchronous or non-deterministic execution within one computation step of a subsystem, depends always on the parent/children priority principle and on the way the subsystem's tasks are attributed (see Section 4.2).

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

The role of attributes is best illustrated by two particular cases (see also the example in Sec. 5.1). The first when all subsystem's tasks are attributed "process" (the system task is attributed "systemprocess") and the second when all of them are attributed "activity" (the system task is attributed "systemactivity"). In the first case *all* (but at most one per task) ready-to-fire transitions (actions) that are not in the ancestor/dependent conflict, are selected (Fig. 5a), while in the second case *only one* of them (Fig. 5b) 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") task may be substructured in both "processes" and "activities" (Fig. 5c).

Although subsystems may exchange messages, they run asynchronously in that their computation steps are completely independent from each other. The relative speed of subsystems are not constrained (synchronized) at all. Recalling the attributing principles of Section 2.2 and the discussion there, this asynchronous behavior is possible since all ancestor (enclosing) tasks of a system task are inactive and nonattributed. Thus, all means of control are absent.

2.5. Global Behavior

To describe the behavior of a complete system specified in Estelle, the operational style (operational semantics) has been used.

The global behavior of a system specified in Estelle is defined by the set of all possible sequences of, so called, "global situations" generated from an initial situation. Two consecutive global situations correspond to execution of one

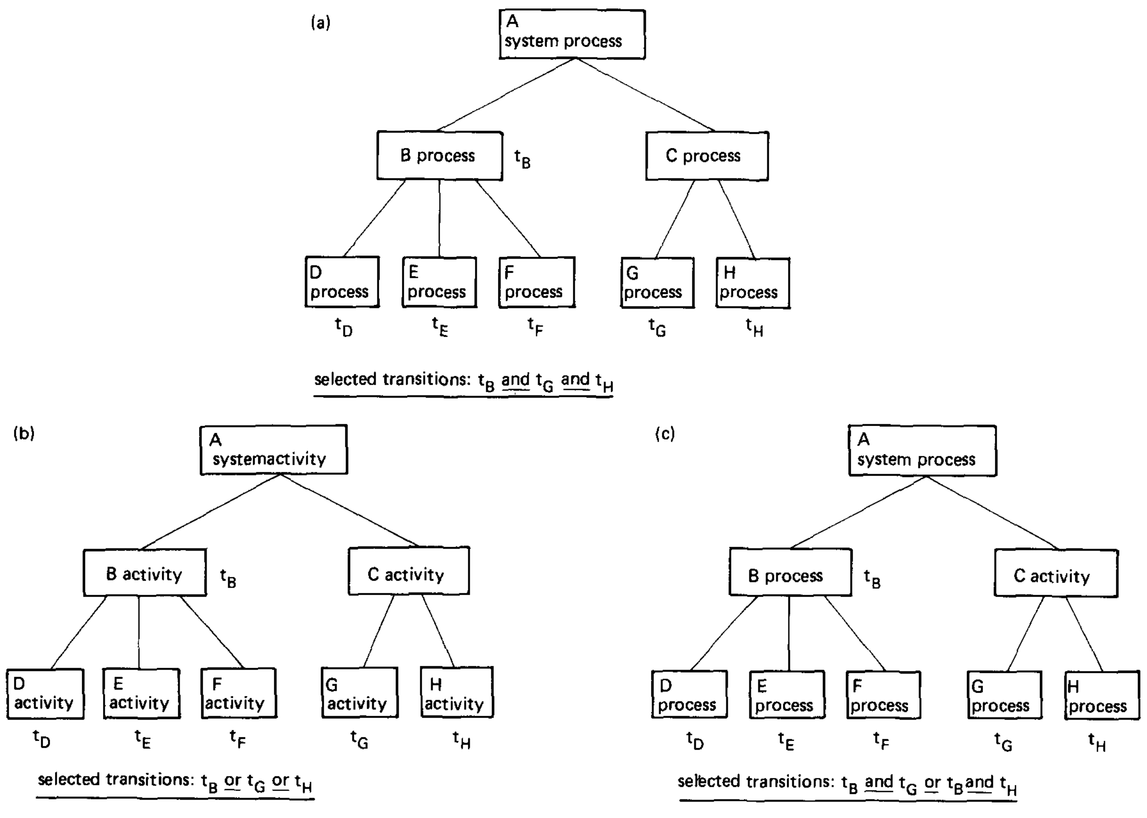


Fig. 5.

transition, i.e., transitions are considered *atomic* in that, conceptually, one cannot observe intermediate results of their execution.

The operational semantics for Estelle describe the way these sequences are generated, i.e., the way the system's transitions (transitions of its tasks) may be interleaved to properly model synchronous parallelism within subsystems combined with asynchronous parallelism between them. These global semantics are described in more detail in Section 4.

It is worth to note that the operational style in defining semantics for Estelle considerably aids in specifying associated tools such as simulator/debugger, an interpreter, and a compiler. It also reinforces a belief of the mutual compatibility of these tools, even if they are designed by different teams.

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 that execution of these transitions must be delayed). However, the retained hypothesis is that execution times of actions are unknown. This knowledge is considered implementation dependent. The computation 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 can not be fired (see Section 3.2.3). The way the "delay-timers" are interpreted is explained, by an example in Section 5.3.

The formulated constraints specify a class of acceptable time processes. In each implementation or for simulation purposes, any element of this class may be chosen (see [9,10]).

3. Highlights on the Syntax of Estelle

3.1. 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 $m1,\ldots, mN$, $n1,\ldots, nK$ 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 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 $m1, m2,\ldots, mN$), and the set of interactions which can be received contains all interactions specified for ROLE2 (i.e., the interactions declared by $n1, n2,\ldots, nK$).

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 connected or linked 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
```

3.2. Modules and Tasks

Tasks are *instances* of *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 tasks 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 task of the module header type is created (initialized). The following is an example of a module header definition:

```
module A systemprocesses (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 are, in principle, 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. "External" bodies nicely serve to allow describing an overall system architecture without detailed description of the system components. This features is illustrated by the example of Section 5.2.

The body definition is composed of three parts:
- declaration part,
- initialization part,
- transition part.

3.2.1. 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. The structure constitutes a textual "pattern" for any dynamic hierarchy of tasks (module instances). Note that a number of tasks may change dynamically, but the hierarchical position of each task corresponds to the position of the task's definition in this pattern. 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. 6.

```
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;
```

Module variables serve as references to module instances (tasks) 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.

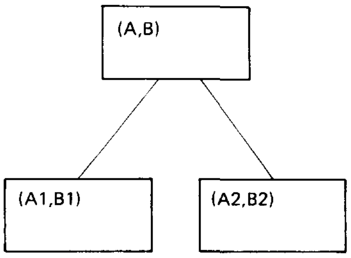


Fig. 6.

A module instance may be created or destroyed by statements referencing module variables (init and release statements, see 3.2.2).

The internal behavior of each module (instance) is defined in terms of a state automaton 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 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 task and its children tasks. They are declared in the same way as the external interaction points within a module header.

3.2.2. 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 tasks can be created. Creation of children tasks during initialization defines their "initial architecture".

To initialize Pascal variables, Pascal statements are used (for example, W := 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 task. 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 (tasks) referenced by the module variables *X*, *Y* and *Z*, respectively. All tasks 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 tasks (referenced by) *X* and *Z* are instances of (A1, B1) and task (referenced by) *Y* is an instance of (A1, B2). The concrete hierarchy of tasks (module instances) of Fig. 7 corresponds to the hierarchical "pattern" of module definitions from Fig. 6.

The initialization also establishes links between appropriate interaction points of the three newly created tasks. These links are also presented in Fig. 7. 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 task replaces that of the parent task).

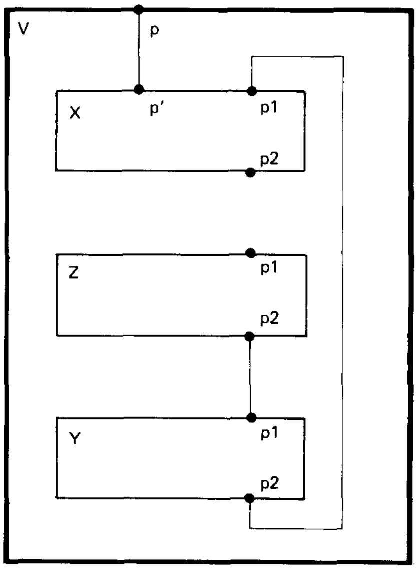


Fig. 7.

3.2.3. 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
    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
    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 $A1, \ldots, An$, 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 once 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 task state if the current value of the task's control variable "state" is among those listed by the from-clause. For example, if IDLE is the current control state of a task, 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 task 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 task state if the boolean expression B evaluates to "true" in that state.

The condition of a transition decides whether the transition is *firable* (or *ready-to-fire*) in a task 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 task will reach a new state.

A transition is said to be *enabled* in a task 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 task state and at a given moment of time if:

(a) it is enabled in the state, and if it is a delay transition, with 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".

The *transition action* is composed of:
- a to-clause (to A, where A is a control state identifier) and
- 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 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 (tasks), 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 reference either an interaction point of the task issuing the statement (i.e., p is an interaction-point-identifier) or an interaction point of one of its children tasks. 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 task currently referenced by the module variable X.

The "release X" statement destroys the task referenced by the module variable *X* and all its descendent tasks.

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 task referenced by *X*.

The "detach *p*" statement detaches the interaction point *p* from the interaction point it was attached to.
(In fact the semantics of both attach and detach statements are slightly more complicated. Roughly speaking, in the case of an attach-statement, the contents of the queue of the referenced interaction point is moved down to the queue of the interaction point at the end of the chain of attachments. In the case of a detach-statement, the contents of the queue of the interaction point at the end of the chain of attachments is moved up to the queue of the referenced interaction point).

If two interaction points *p*l and *p*2 are connected (attached), then the result of "disconnect *p*l" ("detach *p*l") is the same as that of "disconnect *p*2" ("detach *p*2").

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

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

As we explained earlier (see Section 2.3), if *p*l and *p*2 are the two end-points of a communication link, then the "output *p*l.*m*" statement leads to appending interaction *m* in the queue associated with the interaction point *p*2.

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

The restrictions to Pascal [29] adopted in Estelle are mainly the following:
- all declared functions are "pure", i.e., without side effects (but they may return values of an arbitrary type, which is an extension to Pascal),
- 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.

3.3. 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 (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 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 (see Section 5.3 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 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.

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

4. 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.

Recall that the specified system is a collection of *subsystems*, that tasks within each subsystem may execute their transitions in a *synchronous parallel* or *non-deterministic* way, and that tasks belonging to different subsystems may execute their transitions in an *asynchronous parallel* fashion.

The subsystems execute in a succession of computation steps. Each computation step of a subsystem begins by selection of transitions (among those ready-to-fire and 'offered" by the subsystem component tasks) that are to be executed (in parallel) within this computation step. The selected transitions are then executed. A computation step ends when all of them are completed. The execution of transitions within one computation step of a subsystem 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 in 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 proposed model. Nevertheless, the execution of these transitions is synchronized in that a selection of new transitions to execute starts only when all of them have completed.

As opposed to the above, it is assumed that the tasks of different subsystems cooperate in an asynchronous parallel way, therefore their relative speeds are not all constrained.

To properly model the possible behaviors of a complete system specified in Estelle, non-determinism and both forms of parallelisms have to be expressed by the way the transition executions are interleaved. Adequacy of this interleaved model is assured in turn by the assumption of the *atomicity* of transition.

4.1. Global Situations

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

- the hierarchical structure of tasks within the specified system SP, the structure of bindings established between their interaction points, and the local state of each task. 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 ($i = 1, \ldots, n$, where n is the number of subsystems).

Each global situation is denoted by:

$$\text{sit} = (\text{gid(SP)};\ A1, \ldots, An)$$

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 $Ai \neq 0$, the i-th subsystem is executing.

4.2. Next-Situation-Relation

This relation defines the successive situations of an arbitrary current situation

$$(\text{gid(SP)};\ A1, \ldots, Ai, \ldots, An).$$

It is defined in the following manner: For every $i = 1, 2, \ldots, n$,

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

$$(\text{gid(SP)};\ A1, \ldots, AS(\text{gid(SP)}/i), \ldots, An)$$

where $AS(\text{gid(SP)}/i)$ is the set of transitions selected for execution by the i-th subsystem.

(2) If, in the current situation, $Ai \neq 0$, then for each transition t of Ai, the following is a next situation

$$(t(\text{gid(SP)});\ A1, \ldots, Ai - \{t\}, \ldots, An)$$

i.e., the new gid(SP) results from execution of 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).

The execution of a transition t of a task:
- may cause a change in the task's local state. In particular it may create a new child task and/or a new communication link. The transition may also "output", i.e., it may send an interaction which is put into the FIFO queue of another task, etc. All these changes are expressed by t(gid(SP)).
- cannot influence either the choice of transitions already selected by the other subsystems (the sets Aj, for $j \neq i$, remain unchanged in the next situation) or the choice of transitions within the same subsystem (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 subsystem, i.e., the choice of the set AS(gid(SP)/i) is regulated by
- the principle of parent/children priority, and
- the tasks' attributes.

The rule applied to a task within a subsystem can be formulated as follows:
- if the task has a ready-to-fire (fireable) transition to offer, then this one will be selected (parent/children priority),
- otherwise, depending on whether the task is attributed "process" ("systemprocess") or "activity" ("systemactivity"), respectively, all or one (chosen nondeterministically) of those ready-to-fire transitions offered by its children tasks, will be selected.

This rule applied recursively, starting with the root (system) task of the i-th subsystem gives the selected set AS(gid(SP)/i). Fig. 5 illustrates the applications of this rule for two special cases.

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

5. Example

The examples below are not specifications of real systems. They help the reader become familiar with the syntax of Estelle and:
- illustrate how the module attributes influence the global behavior of a specified system (Section 5.1),
- illustrate a way the overall structure of a system can be specified using "external" parameters which replace explicit module body definitions (Section 5.2),
- illustrate and analyze the use and interpretation of the "delay-clauses" in combination with "priority-clauses" (Section 5.3).

5.1. Attributes and Global Behavior

In the specification which we give below, three different ways of assigning attributes are considered. These different assignments are described in Table 1. Syntactically, the particular attributes replace "* * * *" in the specification. Thus, in fact, three distinct specifications are obtained.

```
specification A * * *;
default individual queue;
  const max = 0;
  channel BC(User1, User2);
    by User1: a;
    by User2: b;

module B * * *;
  ip  p1: BC(User1);
end;
body BB for B;
  state Y1, Y2;
  initialize to Y1 begin end;
  trans from Y1 to Y2
     {t1} begin output p1.a end;
  trans from Y1 to Y2
     priority max
       when p1.b
       {t2} begin end;
end;(* end of BB body *)

module C * * *;
  ip p2: BC(User2);
end;
```

```
body CC for C;
  state Z1, Z2, Z3, Z4;
  initialize to Z1 begin end;
  trans from Z1 to Z2
      {s1} begin output p2.b end;
  trans from Z1 to Z3
    priority max
      when p2.z
      {s2} begin end;
  trans from Z2 to Z4
    priority max
      when p2.a
      {s3} begin end;
  trans from Z2 to Z4
      {s4} begin end;
end; (* end of CC body *)

modvar Y:B, Z:C;
initialize
  begin
    init Y with BB;
    init Z with CC;
    connect Y.p1 to Z.p2;
  end;
end.
```

As we stressed earlier in several places, an attribute assigned to a task serves to impose certain kinds of synchronization on the behavior of its descendant tasks. Therefore, modules whose children modules do not exist (bottom level modules in the hierarchy), may be arbitrarily attributed within the general framework of the attributing principles (see Section 2.2). For that reason, in cases 2 and 3, modules B and C may assume any attribute among those mentioned in Table 1 for these cases. A particular choice does not influence the global system behavior (in fact, that is why we have only three really different cases). Observe however, that these modules must be activities in the first case (due to principle (5) in Section 2.2).

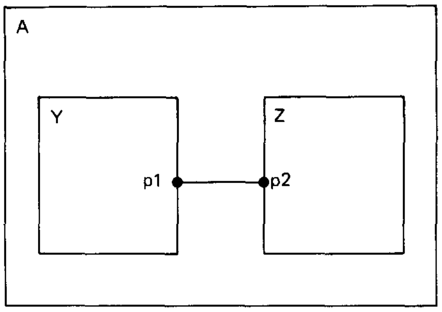


Fig. 8.

The initial and invariant structure of tasks of the specification (as described by its initialization part) is depicted in Fig. 8. Diagrams of the state automata described by modules B and C are shown in Figs. 9a and b, respectively.

Our objective is to show, in this example, how different attributions lead to different admissible sequences of executed transitions. We do not examine all global situations generated by the formal definition of Section 4.2 because they include

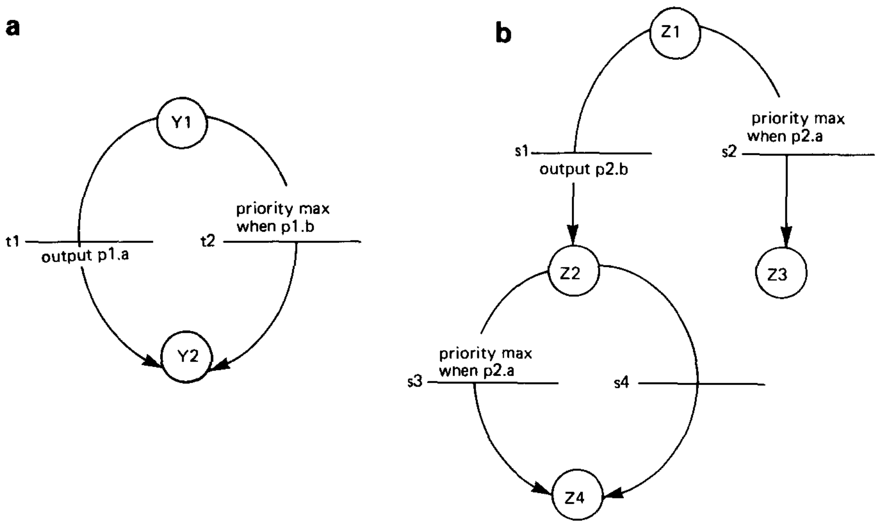


Fig. 9.

Table 1

	1	2	3
specification A	systemactivity	systemprocess	nonattributed
module B	activity	process or activity	systemprocess or systemactivity
module C	activity	process or activity	systemprocess or systemactivity

many "intermediate" situations which serve only to properly define the moment at which a subsystem starts to execute the transitions of its consecutive "computation steps". A pass to such an "intermediate" situation has no effect on the global instantaneous description of a specification (point 1 of the definition of the next-situation-relation in Section 4.2). The transitions in our example have no statement part (with exception of *s*1 and *t*1 which include output statements) because we are not interested what the transitions do but how they may be interleaved.

Let us examine the three cases given in Table 1.

5.1.1. *First Case (1)*

In the first case, the specified system consists of one subsystem (the system itself), hence any of the system's global situations includes exactly one set of "currently executing transitions". Since the system (specification A) is attributed "systemactivity", the set is either empty or consists of one transition (of the task *Y or* of the task *Z*). Therefore, the execution proceeds in a purely nondeterministic fashion.

The computation begins with the initial situation in which:

- *Y* is in its control state *Y*1,
- *Z* is in its control state *Z*1,
- *p*1-queue and *p*2-queue are empty, and
- the set of "currently executing transitions" is empty.

The system A may now choose for execution either transition *t*1 of *Y* or *s*1 of *Z*. If *t*1 is chosen and executed, then the task *Y* ends its computation in its control state *Y*2, the message "a" is at the head of the *p*2-queue, and *s*2 is the only possible transition to execute in *Z* (priority). The system chooses and executes *s*2 (removing "a" from the *p*2-queue) and the whole computation of the system *A* ends. The sequence ⟨*t*1, *s*2⟩ of transitions has been executed.

Suppose now, that in the above first step the transition *s*1 of *Z* is chosen and executed. Then the message "b" is at the head of the *p*1-queue and *t*2 is the only transition which may be now executed in the task *Y* (priority). On the other hand, the task *Z* is in its control state *Z*2 and it may only execute *s*4. Whatever of the two transitions is chosen by the system to execute first, the other remains ready to be executed next (and last). Therefore, by the choice of *s*1 in the first step, both sequences ⟨*s*1, *t*2, *s*4⟩ and ⟨*s*1, *s*4, *t*2⟩ are admissible,.

Summarizing, in the case (1), the set of admissible transition sequences consists of

⟨*t*1, *s*2⟩, ⟨*s*1, *t*2, *s*4⟩ and ⟨*s*1, *s*4, *t*2⟩.

5.1.2. *Second Case (2)*

In the second case the specified system also consists of only *one* subsystem (the system itself), hence as in (1), there is exactly one set of "currently executing transitions" in any of the system's global situations. However, because the system (specification A) is attributed "systemprocess", the set may have 0, 1 or 2 elements, i.e., the set is either empty or it consists of at most one transition of the task *Y and* at most one transition of the task *Z* (if there are two transitions in this set, then their executions are synchronized). The initial situation is the same as that described in (1).

The system-process A chooses one ready-to-fire transition of the task *Y and* one such transition of the task *Z* (the set *AS* in Section 4.2) for synchronized execution in the system computation step. In the initial situation the only possible choice is: the transition *t*1 of *Y* and *s*1 of *Z*. The transitions are executed in an arbitrary order ending the first computation step of the system A. After this step the task *Y* ended its execution in its control state *Y*2 (i.e., the task has no transition to execute from this state), and *Z* is in its control state *Z*2. The message "a" is at the head of the *p*2-queue. Thus, *s*3 is the only "ready-to-fire" transition (priority) and this transition is executed in the next and last computation step of the system A.

In case (2) of our example, the set of admissible transition sequences consists of two sequences

⟨*t*1, *s*1, *s*3⟩ and ⟨*s*1, *t*1, *s*3⟩.

5.1.3. *Third Case (3)*

In the third case, the specified system consists of *two* subsystems, namely, tasks Y and Z (therefore executions of transitions in Y and Z are asynchronous). According to the definition in Section 4.1, there are exactly *two* sets of "currently executing transitions" in any of the system global situation: one for the subsystem Y and one for Z. Since, in this simple example both subsystems are reduced to one task, each of the sets contains at most one transition (of Y and Z, respectively).

Let us start with the initial global situation whose first three components are given in (1) and in which both sets of "currently executing transitions" are empty. Because the relative speeds of subsystems are not controlled, the following three subcases may happen:

(a) Subsystem (task) Y began and ended to execute $t1$ (the only ready-to-fire transition of Y in the initial situation) before subsystem Z even started. In such a case, subsystem (task) Z is obliged to execute $s2$ (priority) because the message "a" is already in the $p2$-queue. The overall computation ends by executing $\langle t1, s2\rangle$.

(b) This case is symmetric to (a). Subsystem (task) Z began and ended to execute $s1$ (the only ready-to-fire transition of Z in the initial situation) before subsystem Y even started. In such a case we achieve a global situation in which subsystem (task) Y is obliged to execute $t2$ (priority) because the message "b" is already in the $p1$-queue, and subsystem Z can start to execute $s4$. Independently which one of transitions $t2$ and $s4$ began earlier, either one may complete first. Hence, in this subcase, the two sequences of transitions $\langle s1, t2, s4\rangle$ and $\langle s1, s4, t2\rangle$ are admissible.

(c) Suppose now that both subsystems Y and Z independently started to execute $t1$ and $s1$, respectively. Recall that a transition once started must complete and that the transition "intermediate" results can only be observed after the transition is completely executed (atomicity of transitions).

Subsystem Y may complete the execution of $t1$ first and, in such a case, subsystem Z is forced to execute $s3$ (priority) after $s1$. This gives the sequence $\langle t1, s1, s3\rangle$.

If execution of $s1$ ended before $t1$, then Z may begin (and even end) the execution of $s4$ before $t1$ completed. This leads to two admissible sequences $\langle s1, s4, t1\rangle$ and $\langle s1, t1, s4\rangle$.

It may happen, however, that $s1$ completed first but subsystem Z did not start its consecutive execution before the end of execution of $t1$ (Z was still in its "system management phase" when $t1$ ended). In such a case, Z can only execute $s3$ next (priority) and this leads to the admissible sequence $\langle s1, t1, s3\rangle$.

Summarizing, in case (3), the set of admissible execution sequences consists of:
$\langle t1, s2\rangle$, $\langle s1, s4, t2\rangle$, $\langle s1, t2, s4\rangle$, $\langle t1, s1, s3\rangle$, $\langle s1, s4, t1\rangle$, $\langle s1, t1, s4\rangle$ and $\langle s1, t1, s3\rangle$.

From the above example one can derive some interesting and general properties concerning behaviors of Estelle specifications:

● The specifications which consist of one subsystem and which differ only by the system attribute, may lead to different executions (cases (1) and (2) above). In general, this means that the attributes "activity" and "process" play really different roles (the prefix "system" serves only to distinguish the "top" level of the attributed modules within a specification).

● The execution sequences of (1) and (2) are both included in those of (3). This justifies the use of the term "synchronous execution of tasks Y and Z" in cases (1), and (2), with respect to completely independent, hence "asynchronous", execution of tasks Y and Z in case (3). As a matter of fact, "synchronous" in case (1) means simply "non-deterministic". Note also, that asynchronous behavior may produce execution sequences which do not appear in both synchronous behaviors ($\langle s1, s4, t1\rangle$ and $\langle s1, t1, s4\rangle$ in the example).

● It is interesting to compare the three sets of execution sequences for the same three assignments of attributes (cases (1)–(3)) for the above specification in which all priority-clauses were removed. In such an example the set of execution sequences in case (2) is, as before, a proper subset of the set of those sequences in case (3). The last set, however, equals the set of transition sequences generated in case (1) (of course, all sets are different than those of the above example).

The above property concerning equality of execution sequences in cases (1) and (3), is valid for any Estelle specification in which priority-clauses do not appear *and* in which all system modules are at the bottom level of the hierarchy of modules (all other modules are inactive and not attributed), i.e., the set of execution sequences remains the same if, in a specification satisfying the above constraints, one changed module attributes, assigning: "systemactivity" to the specification module, and "activity" to all other modules.

The property confirms the known fact that, in some cases, asynchronous parallelism can adequately be modelled by nondeterminism. It simplifies the model [10] and its use (e.g., for verification purposes as in [21,22,34]). It must be noted, however, that the above restriction imposed on the Estelle specifications not only exclude priorities among a module's transitions but also constraint the attractive Estelle possibilities of dynamic creation and destruction of tasks and their communication links.

5.2. 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. While the RECEIVER_BODY is further specified in Section 5.3 as one simple module body (without substructuring), the other subsystems remain "external" and their definitions can be differently substructured in subsequent refinements of the specification. The scheme of the specified EXAMPLE system is presented in Fig. 10.

```
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
```

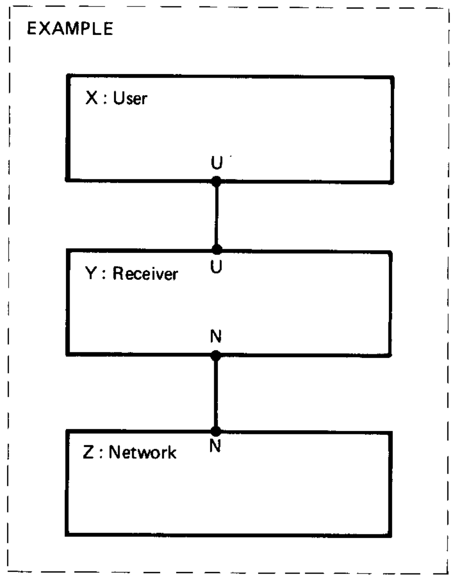


Fig. 10.

```
  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.
```

5.3. Interpretation of Delay-Clauses

The computational model for Estelle is formulated, as far as possible, in time-independent terms. But 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($E1$, $E1$)".

The intuitive interpretation above of the delay-clauses is illustrated below by an example (taken from [19] with slight changes). 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 Section 5.2. The necessary channel definitions and declaration of interaction points are then defined at the specification level of the EXAMPLE in Section 5.2.

The following narrative description assumes a protocol in which:
- multiple protocol data units (messages) may be acknowledged in a single acknowledgment;
- each protocol data unit shall be acknowledged after some maximum time (max);
- it is desirable to acknowledged each protocol data unit received after a minimum time (min), but when more than four are received an acknowledgment must be sent after this minimum delay (min);
- when the system remains inactive too long (inactive period), "dummy" acknowledgements are generated to provide minimal activity to prevent disconnections.

The following module body describes a solution in Estelle. The specification in Section 5.2 assumes a time scale resolution of 1 second (see "timescale" option in the specification).

```
body RECEIVER_BODY for RECEIVER;
type                        (* declaration part *)
  time_period = integer;
const
  high = 0;
  medium = 1;
  low = 2;
state
  IDLE, AK_SENT;
var
  ak_no: integer;
  min, max, inactive_period: time_period;
initialize                  (* initialization part *)
  to IDLE
    begin
    min := 1;
    max := 20;
    inactive_period := 60;
    ak_no := 0;
  end;
trans                       (* transition part *)
  from IDLE
    to IDLE
      priority medium
        when N.DATA_INDICATION;
          {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)
          {t2} begin
              output N.SEND_AK(ak_no)
              end;
    provided (ak_no > 4) and (ak_no < 7)
      priority high
        delay(min)
          {t3} begin
              output N.SEND_AK(ak_no)
              end;
    provided ak_no = 7
      priority high
          {t4} begin
              output N.SEND_AK(ak_no)
              end;
    provided ak_no = 0
      priority low
        delay (inactive_period)
          {t5} begin
              output N.SEND_AK(ak_no)
              end;
from AK_SENT
  to IDLE
          {t6} begin
              ak_no := 0
              end;
end;
```

The above transition part of the RECEIVER_BODY consists of one nested transition representing in fact six simple transitions. The outer most level of factorization (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 simple transitions $t2$–$t5$. These transitions in turn, are factorized by their provided-clauses.

From the initialization part, we see that the module begins in its IDLE control state, with the delay values appropriately initialized and the number of messages that remain to be acknowledged, set to 0. Note that immediately after the initialization, 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 increase by 1 (transition $t1$). In this case the timer of transition is canceled. Otherwise, i.e., when 60 seconds passes without any incoming message, an "artificial" acknowledgment is sent to prevent disconnection (transition $t5$).

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), when 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 (the transition becomes no longer enabled), then its timer is canceled.

Immediately after the 5-th 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 7-th message is served (transition *t1*) during this period, the transition will not be executed and its timer will be canceled. The transition *t4* will be executed instead.

It is worth noting the role which is played by priorities assigned to transitions. Due to them, in a heavy traffic of arriving messages (more than 7 in less than one second), acknowledgments 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 acknowledgments are sent always for blocks of 5–6 messages. If the messages arrive in quantities 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.

6. Conclusion

An overview of the most important concepts and facilities of Estelle has been presented. However, there still are aspects of the language which have been only mentioned or even omitted. For example, we only mentioned that the result of executing an "attach" or "detach" statement may have some effect on contents of queues of the interaction points being attached or detached, respectively. The details are left unexplained (they become quite complicated when these statements reference an interaction point whose queue is shared with other interaction points). We have not considered special Estelle constructs "all", "for one" and "exist" which enrich the Pascal part of the language, and we have not introduced the any-clause which (together with nesting) provides a means for more compact description of transitions.

The presented examples neither illustrate all introduced Estelle mechanisms nor do they pretend to give very practical guidelines for writing Estelle specifications of real systems and/or protocols. The readers interested in real-life application examples may examine several ISO protocols and services which have been described in Estelle (e.g., Protocol for providing the connectionless-mode network service, Network Service, Transport Protocol and Service, Session Protocol and Service, Presentation Protocol and Service, FTAM-SPAG profile Protocol, Virtual Terminal Protocol etc., see [23,25,26,27], and overview papers [19] and [28]). It has to be noted that some of these descriptions used earlier versions of Estelle. Nevertheless, they constitute quite rich material for further experiments.

It is appropriate to end this presentation emphasizing again the importance of the Estelle supporting tools. Given the size of existing Estelle specifications, it seems clear that without such tools comparable specifications cannot efficiently be developed and reliably implemented. Some of these tools already exist and are tested. Other are being designed. 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., [7,9,11,13–15,17,21,22,28, 30,33,34]).

Acknowledgments

The authors wish to thank all our colleagues for their helpful critique and comments. Special thanks are due to P. Amer, V. Chari, J-P. Courtiat, J. Linn, and R. Tenney who have contributed many useful suggestions and improvements.

References

[1] Ansart J.P., Rafiq O., Chari V., Protocol Description and Implementation Language (PDIL), Proceedings IFIP 2nd International Workshop on Protocol Specification, Verification and Testing, C. Sunshine (ed), North Holland, 1982.

[2] Ayache J.M., Courtiat J.P., Diaz M;, LC/1A Specification and Implementation Language for Protocols, Proc IFIP 3rd International 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 environment logiciel pour la conception des systèmes distribués, Proc. 3ème Congrès "De nouvelles architectures pour les communications", Paris 28-30 October 1986.

[7] Courtiat J.P., Dembinski P., Groz R., Jard C., ESTELLE: un language pour les algorithmes distribués et les protocoles. Technique et Science Informatiques, vol. 6, No. 2, 1987.

[8] Dembinski P., Estelle Semantics, SEDOS Rep. SEDOS/054, June 1986.

[9] Dembinski P., Budkowski S., Simulation Estelle Specifications with Time Parameters, Proc. IFIP WG 6.1 Seventh International 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.

[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] Ansart J.P., Chari V., Dembinski P., De la Spécification à l'Implementation en Estelle, 9ème Journée Francophone sur l'Informatique, Liège, 20–21 Janv., 1987.

[13] User Guide for the NBS Prototype Compiler for Estelle, Rep. No. ICST/APM 87-1, NBS Institute for Computer Science and Technology, 1986.

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

[15] Bochman G., Usage of Protocol Development Tools: the results of a survey, Proc. IFIP WG 6.1 Seventh International Conference on Protocol Specification, Testing, and Verification (eds. H. Rudin and C.H. West), North-Holland 1987.

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

[17] Kato T., Hasegara T., Horinchi H., Design of Translator from Estelle with ANS.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 language 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 International 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, CIS Dept. Tech. Report 87-10, University of Delaware, August 1987.

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

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

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

[28] Diaz M., Vissers Ch., Budkowski S., Estelle and LOTOS Software Environments for the Design of Open Distributed Systems, Proc. ESPRIT Conference Week, Brussels, Spet. 28-30, 1987.

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

[30] User Guide for the BULL Prototype Compiler for Estelle, BULL, S.A., Corporate Networking and Communication (DRCG), Distributed System Architecture and Standards (ARS), LOUVECIENNES, 1987.

[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, November 1987.

[33] De Saqui-Sannes P., Courtiat J-P., ESTIM: An Interpreter for the Simulation of Estelle Descriptions, SEDOS Rep. SEDOS/115, November 1987.

[34] Parapanagiotikais G., Azema P., Shezalviel-Pradin B., On a Prolog Environment for Protocol Analysis, Proc. International Conference on Distributed Computing Systems, Cambridge Ma., May 1986.