INTERNATIONAL STANDARD

ISO/IEC 9074

Second edition
1997-12-15

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

Technologies de l'information — Interconnexion de systèmes ouverts (OSI) — Estelle: Technique de description formelle basée sur un modèle de transition d'état étendu

Reference number
ISO/IEC 9074:1997(E)

Contents

Foreword

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

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

International Standard ISO/IEC 9074 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 21, *Open systems interconnection, data management and open distributed processing*.

This second edition cancels and replaces the first edition (ISO 9074:1989), which has been technically revised. It also incorporates Amendment 1:1993.

Annex C forms an integral part of this International Standard. Annexes A, B and D are for information only.

0 Introduction

0.1 General

Formal description techniques (FDTs) are methods of defining the behavior of an (information processing) system without using a natural language such as English. In the following sub-clauses of this introduction, the importance of FDTs and their standardization is discussed. The objectives that an FDT must satisfy are considered.

This International Standard describes an FDT for the specification of communication protocols and services. The FDT is based on an extended finite state transition model.

This International Standard addresses two different groups of people: protocol designers, who may use this FDT to specify protocols; and protocol implementers, who may implement a protocol described using this FDT, or alternatively implement a tool to process such a protocol description automatically. Implementers must ensure that they follow the semantics of the FDT, as described in this International Standard. Protocol designers must ensure that their specification relies on the characteristics of an implementation only to the extent that the semantics of the FDT is defined by this International Standard.

This International Standard comprises nine clauses and four annexes. Annex A, Annex B, and Annex D are informative and not an integral part of this International Standard; Annex C is normative and is an integral part of this International Standard.

0.2 Formal Description Techniques

Formal description techniques are important tools for the design, analysis and specification of information processing systems. It is by means of formal techniques that system descriptions can be produced that are complete, consistent, precise, concise and unambiguous. This is only possible if an FDT is self-contained, so that the descriptions given in an FDT need not refer to any informal knowledge of the system that is described. An important aspect of a formal system is that it allows analysis by formal methods. An FDT that has such a formal basis can be used to demonstrate the correctness of a specification.

The specifications and descriptions using this FDT are intended to be formal in the sense that it is possible to analyze and interpret them unambiguously. The FDT supports a structuring of a specified system into smaller parts and the specification of the behavior of a part in terms of an extended finite state transition model.

0.3 Requirement for standard FDTs

If an FDT is defined in an International Standard, the description is available to all who require it. The directives for the production of such a standard require a high degree of international acceptance and technical stability. Any amendment also requires international agreement. Hence a standard FDT offers the most useful form of presentation to those who wish to apply it.

0.4 Objectives to be satisfied by an FDT

Although this International Standard describes an FDT that is generally acceptable to distributed, concurrent information systems, it has been developed particularly for OSI. The main objectives to be satisfied by such an FDT are that it should be:

— expressive: an FDT should be able to define both the protocol specifications and the service definitions of the seven layers of OSI described in ISO/IEC 7498-1.

— well-defined: an FDT should have a formal model that is suitable for the verification of these specifications and definitions. The same model should support the validation of implementations that are permitted by the OSI International Standards. This model should also support the testing of an implementation for conformance.

— well-structured: an FDT should offer means for structuring the description of a specification or definition in manner that is meaningful and intuitively pleasing. A good structure increases the readability, understandability, flexibility, analyzability and maintainability of system descriptions.

— abstract: there are two aspects of abstraction that an FDT should offer:

— an FDT should be completely independent of methods of implementation, so that the technique itself does not provide any undue constraints on implementers.

— an FDT should offer the means of abstraction from irrelevant details with respect to the context at any point in a description. Abstraction can reduce the local complexity of system descriptions considerably. In the presence of a good structure, abstraction can help even further to reduce the complexity of descriptions.

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

1 Scope

This International Standard defines the semantics and syntax of the Formal Description Technique Estelle. Estelle is in general used for the formal description of distributed, concurrent information processing systems. In particular Estelle can be used formally to describe the service definitions and protocol specifications of the layers of Open Systems Interconnection described in ISO/IEC 7498-1. This International Standard does not define methods for the verification of specifications written in Estelle.

2 Normative References

The following standards contain provisions which, through reference in this text, constitute provisions of this International Standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO/IEC 7185:1990, *Information technology — Programming languages — Pascal.*

ISO/IEC 7498-1:1994, *Information technology — Open Systems Interconnection — Basic Reference Model: The Basic Model.*

ISO/IEC 646:1991, *Information technology — ISO 7-bit coded character set for information interchange.*

3 Definitions

For the purposes of this International Standard, the following definitions apply.

3.1 denotation: A mathematical construction designated according to the requirements of this International Standard as the meaning (interpretation) of an Estelle specification or part of such a specification.

3.2 undefined: A property of an Estelle specification or part of such a specification such that it has no denotation according to the requirements of this International Standard.

3.3 error: An Estelle specification that does not conform to the requirements of this International Standard is in error. Its meaning cannot be determined by applying the semantics of this International Standard.

4 Conformance

A formal specification written in Estelle conforms to the requirements of this International Standard if and only if it is derivable according to the syntactic rules and constraints given in clauses 7, 8 and Annex C, and it has a denotation according to the semantic rules given in 5.3 and clause 9.

Paragraphs labelled NOTE and **Informal Semantics** are included to aid the reading of this International Standard, but are not part of this International Standard for purposes of determining conformance. These paragraphs often contain information that may be derived from other parts of the International Standard, and thus a specification that conforms to this International Standard will satisfy the note. Paragraphs labelled **Remark** are a normative part of this International Standard.

Remark: A formal specification expressed in Estelle may also be used as the basis for an implementation of a system conforming to an International Standard for Open Systems Interconnection. In such cases, it may be processed by a compiler or other computer-based software tool. This International Standard does not, however, specify requirements for such compilers or tools.

5 The model

This clause describes the basic concepts of a model for defining the semantics of an Estelle specification. Details on syntax and semantics can be found in clauses 7 and 9, respectively.

A specification describes a hierarchically structured system of nondeterministic sequential components (instances of *modules*) interchanging messages (called *interactions*) through bidirectional links between their ports (called *interaction points*). Both the hierarchy of modules and the structure of links may change over time, thereby making the system a dynamic one.

This clause is organized into three parts.

— First (5.1) a single *module instance* is briefly characterized, together with the role it plays with respect to its directly subordinated modules (called *children instances*). The module instance definition is the main part of the Estelle semantics in clause 9.

— Second (5.2) a set of nesting principles and their impact on the degree and character of parallelism and nondeterminism between module instances are described. The importance of classifying a module into a **process**, **activity**, **systemprocess**, or **systemactivity** is explained in this context.

— Third (5.3) the system behavior, as given by an Estelle specification, is defined in terms of transformations of a system *global situation* which includes the existing (i.e., instantaneous) hierarchy of instances of modules in their local states and the current structure of links between them. An execution of an action (called a *transition*) by a module instance causes an observable change in the global situation. Although an action may change several elements of this situation, these changes are observable only in their entirety. This reflects a central principle of Estelle that the execution of a transition is atomic. The way that actions are interleaved characterizes possible sequences of system computations. These computations, in turn, model the results of admissible real executions.

A separate subclause (5.3.5) explains the role of time with respect to the progress of a specification's execution; i.e., it describes general assumptions with respect to which, so called, "delays" are treated in the model.

5.1 Module instance

In Estelle a module is defined by a **module-header-definition** and a **module-body-definition** associated with this header. There may be more than one **module-body-definition** referencing the same **module-header-definition**. Thus each pair, a **module-header-definition** and an associated **module-body-definition**, constitutes one Estelle module definition, or more simply, module. Several instances of a module may be created and simultaneously present during execution of an Estelle specification. Each one of them has the same external visibility characterized by the **module-header-definition** and the same internal behavior defined by the **module-body-definition**.

From an external viewpoint, a module instance is a "black box". Access in and out of that box is made with finite sets of interaction points and *exported variables*. These are the only means of communication with the enclosing environment defined by the parent module instance.

Each interaction point of a module instance has an associated first-in first-out (FIFO) queue which receives and stores interactions sent to that module instance through this interaction point. More than one interaction point of a module instance may share the same queue. A module instance may also send, i.e., output, interactions to other modules (or even itself) through its own interaction points. The sets of interactions which may validly be received and sent through a specific interaction point are determined by the **channel-definition** with which the interaction point is declared in the **module-header-definition**.

Exported variables are also declared in the **module-header-definition**. External access (read/write) of exported variables, however, is restricted exclusively to the parent module instance.

The visible results of a module instance's behavior, in the form of the output interactions it sends and the changing values of its exported variables, are the effect of a module instance's internal activity as described within an associated **module-body-definition**. The internal behavior of a single module instance may be characterized in terms of a nondeterministic state transition system, i.e., by determining the system's: set of states, subset of initial states and next-state relation. How these elements are represented in Estelle is briefly described below.

Each *state* has a complex structure which consists of the following items:

— *Control-part* represented by a value associated with the special identifier STATE. The set of *control values* is always finite (hence one often uses the term *finite (control) state transition system.*)

— *Input-environment-part* represented by the contents of all queues associated with a module's interaction points.

— *Internal-data-part* represented by values of variables and parameters occurring within a module definition.

— *Interconnection-hierarchy-part* represented by

(a) the current set of submodules (children instances), and

(b) the current link structure of the module instance's interaction points and its children instances' interaction points (i.e., the connect-attach structure in the state).

The manipulation of this last part of a state, that is, the creation and destruction of children instances and the modification of interaction point links constitutes an important aspect of Estelle.

Initial states of a module instance are defined by a module **initialization-part**. This part of a **module-body-definition** describes an initial control state, the way variables are initialized, and defines an initial hierarchy and interconnection structure of descendant module instances, if any exist.

The *next-state relation* or one-to-many *next-state function* (recall that the state transition systems are, in general, nondeterministic) of a module instance is specified within the transition part of a **module-body-definition** by transitions. Each transition is composed syntactically of two parts: a **clause-group** and a **transition-block**.

— A **clause-group**, which is a set of zero or more clauses, determines (among other things), for each state, whether a transition is *enabled* in this state, i.e., whether certain conditions are satisfied, thereby enabling (but not necessarily requiring) its execution. The clauses also define the control states from which a transition may take place and specify the next control state following the transition's execution.

— The interpretation of a **transition-block** defines an action or operation to be executed by the transition. The action may change the module instance state described above (a change of control state is given by some clauses) and may output interactions to the environment. A transition block is given as a Pascal-like compound statement with some specific restrictions and extensions of Estelle.

Execution of a transition by a module instance is considered to be an atomic operation. Once a transition's execution is started, it cannot be interrupted, and conceptually one cannot view intermediate results regardless of how "large" a transition is (i.e., how many statements it has). The atomicity of a transition is an important assumption for expressing the behavior of a specification in abstract terms, independent of the precise way that atomic transitions are implemented. Atomicity also has an impact on the results of parallel execution of module instances.

A module (and any instance of a module) with an empty **transition-part** in its **module-body-definition** (i.e., a module with no transitions specified), is called *inactive*. A module (and any instance of a module) with a nonempty **transition-part** is called *active*. These notions are used and further explained in the next subclause. An inactive module instance, once initialized, performs no action, although its descendent modules certainly may.

5.2 Nesting of modules and parallelism

The **declaration-part** of a **module-body-definition** may nest (i.e., include) other module definitions (called *children*) which may in turn include other module definitions, and so on. The way that existing module instances of a specification behave with respect to each other (i.e., the degree and form of parallelism and/or nondeterminism involved) is strictly dependent on the textual nesting of the module definitions and the way their headers are qualified by keywords: **systemprocess**, **systemactivity**, **process** and **activity**. This relative behavior is strongly regulated by another Estelle principle, that of execution priority of a transition of a parent module instance over transitions of its children.

The nesting principles and their consequences are formulated in 5.2.1. The role they play in defining what actions may occur in parallel and what level of nondeterminism exists in a specification is explained in 5.2.2.

Within this International Standard, the terms *process module*, *activity module*, *systemprocess module*, and *systemactivity module* will be used to distinguish between module definition categories. These are called *attributed* modules. Similarly the words *process*, *activity*, *systemprocess*, or *systemactivity* will mean **process**, **activity**, **systemprocess**, or **systemactivity** module instance, respectively. In case it is not important whether or not a module is a **systemprocess** or **systemactivity**, the terms *system module* and *system* are used for a module and an instance of that module, respectively.

5.2.1 Nesting of modules

There are the following five nesting principles for defining modules within modules:

(a) Every active module must be attributed; i.e., if for a given **module-header-definition** there is at least one **module-body-definition** with its transition-part nonempty, then the **module-header-definition** must include one of the four keywords: **process**, **activity**, **systemprocess** or **systemactivity**.

(b) A system module cannot be nested (i.e., embodied) within an attributed module. As a consequence all modules embodying system modules must be inactive.

(c) Process and activity modules must be nested within a system module.

(d) A process or systemprocess module may be substructured only into process or activity modules; i.e., descendent (embodied) modules of a process or systemprocess module must be attributed with the keyword **process** or **activity**. Together with (b) and (e), this means that non-attributed modules are only those inactive modules embodying system modules, if such modules exist.

(e) An activity or systemactivity module may be substructured only into activity modules.

In addition to these static principles, it is important to stress the dynamic constraint that a module instance may be created and released or terminated exclusively by the instance whose **module-body-definition** directly includes its definition (i.e., the parent module instance). The same is true as regards the creation and destruction of interaction point links.

This dynamic constraint together with (a) through (e) have the following important consequences:

— A dynamic hierarchy of module instances has a static textual pattern in an Estelle specification. Although the number of instances of a specific module may change in that dynamic structure, the hierarchical position of each instance corresponds to the respective position of its module definition in the textual pattern.

— From the principles (a) through (e) there is exactly one level of system module along any path within the hierarchy of module instances within a specification (except in the degenerate case, where all specification modules are inactive and consequently may remain unattributed, there may be no system level) and, by (b), all embodying modules of a system are inactive. Consequently, any initial state of a specification defines a fixed number of system instances and interaction point links above them. This structure once initialized cannot change during the life-time of the specified configuration since parent modules are inactive; it is invariant. A specification itself will reduce to a single system module if it is attributed (and it must be attributed if it is active).

Example

Figure 1(a) shows an example of a textual pattern of a hierarchy structure of a specification. Figures 1(b) and 1(c) show two possible initializations of that structure.

Figure 1(b) shows only system modules instantiated with one instance of systemprocess module A, one instance of systemactivity module B1, and two instances of systemprocess module B2. Internal substructuring of these systems, if any, may take place later in an execution, assuming the system modules are active. Note that, after the initialization process, the bold line system module instances may not disappear, and also that new ones cannot be created.

Figure 1(c) shows another possible initialization with more internal system substructuring. This internal structure may later be changed dynamically. Note that systemprocess module B2 was not instantiated in this example and will therefore never be used, as module B is inactive. Possible links between interaction points which could have been created during the initialization are not considered in this example.

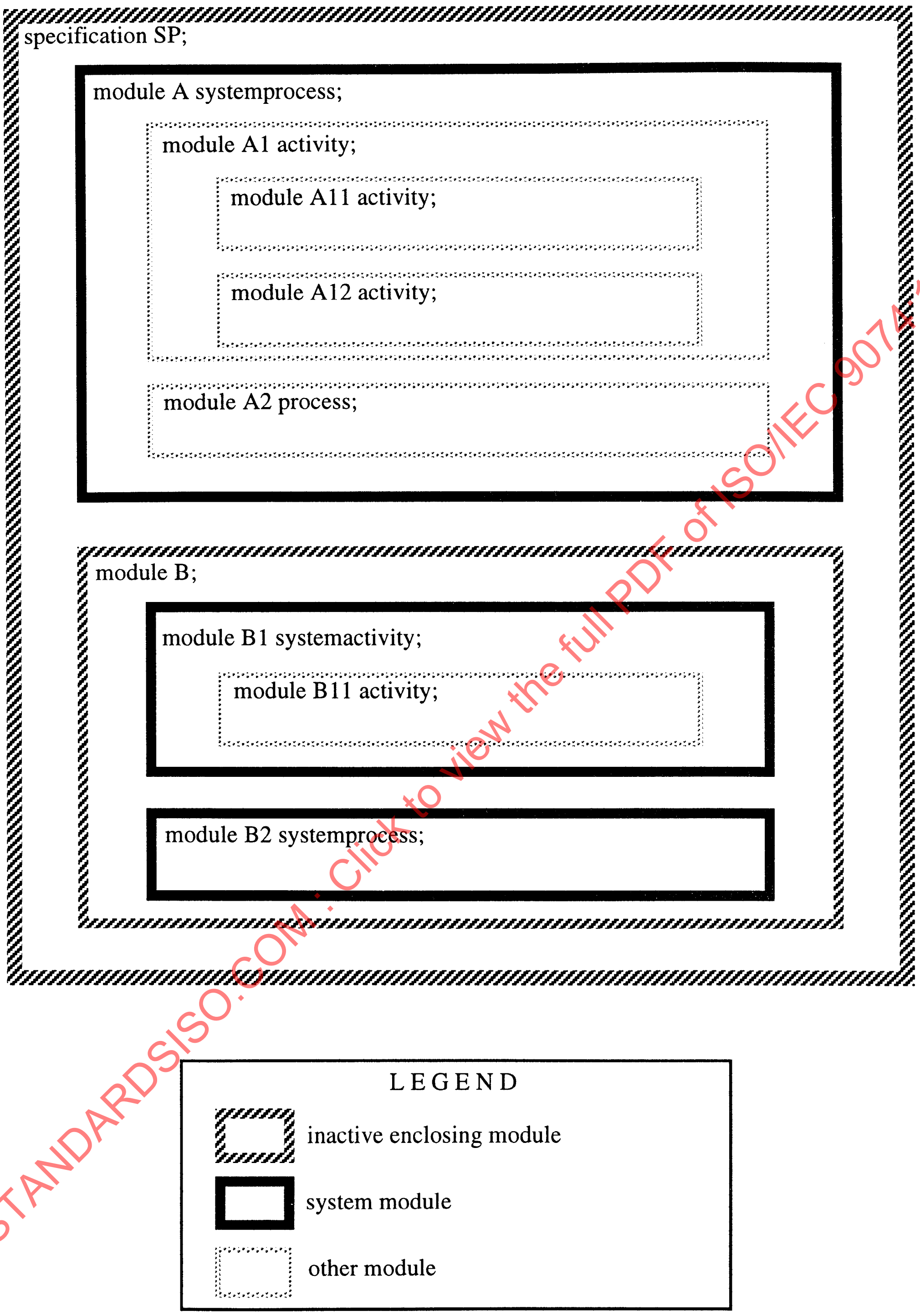

specification SP;
module A systemprocess;
module A1 activity;
module A11 activity;
module A12 activity;
module A2 process;
module B;
module B1 systemactivity;
module B11 activity;
module B2 systemprocess;
LEGEND
inactive enclosing module
system module
other module

Figure 1(a)

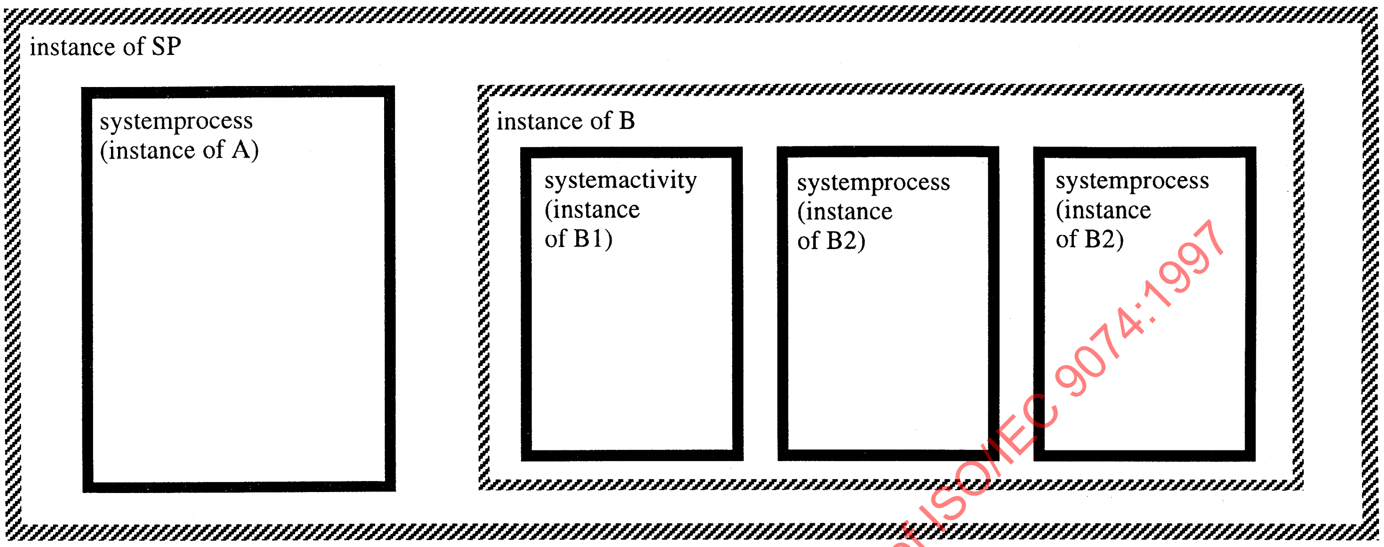

instance of SP
systemprocess (instance of A)
instance of B
systemactivity (instance of B1)
systemprocess (instance of B2)
systemprocess (instance of B2)

Figure 1(b)

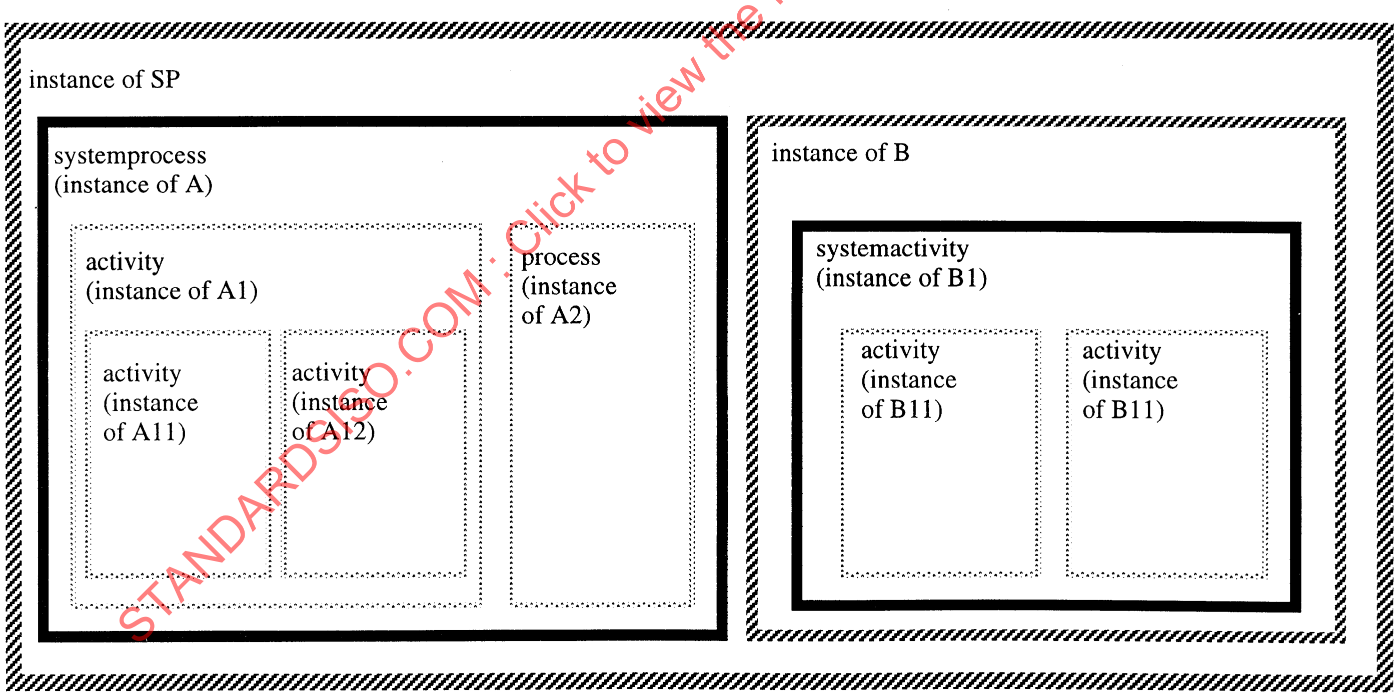

instance of SP
systemprocess (instance of A)
activity (instance of A1)
activity (instance of A11)
activity (instance of A12)
process (instance of A2)
instance of B
systemactivity (instance of B1)
activity (instance of B11)
activity (instance of B11)

Figure 1(c)

5.2.2 Parallelism and nondeterminism

As noted above, only an active module instance may dynamically create, release, and change the connection configuration of its children instances; in doing this, an active instance acts as a supervising-like manager over its children instances. For this and other reasons, such as prevention of race conditions, transitions of a parent instance have priority over its children's transitions. While a module instance transition is executed, all children transition executions are suspended. This property means that a parent instance supervises (or synchronizes) the execution of its descendants. Intuitively speaking, each time a transition in a child module is ready to fire (or, in terminology introduced and used later on, each time a child module offers a transition for execution), it must first "ask" its parent for permission. A parent may decide whether or not to grant that permission only if children transitions that were previously granted permission have completed. This priority relation is transitive, meaning that a module instance transition has priority over all of its descendant instances' transitions. This excludes any parallelism among instances which are in an ancestor/descendant relationship.

From 5.2.1, it follows that system modules are not included in any active module and, therefore, they (i.e., their instances) are not "supervised" by means of a parent module (instance) priority. Hence systems behave fully asynchronously with respect to each other. From this point of view there is no difference between a systemprocess and a systemactivity module. They are differentiated by their internal behavior as explained below. The parallelism within one system (i.e., among instances of descendant modules of a system module) is regulated by a parent/children priority principle and by process/activity attributes that must be assigned to each module definition nested within a system module.

The intention of using the attribute **process** or **activity** for a module (and its instances) is to distinguish two possible forms of execution. These two forms represent a synchronous parallel execution (parallel but synchronized by the parent/children priority), and nondeterministic sequential execution (also preserving the priority principle), respectively (see 5.3.3 for the exact formulation of the behavioral model). This explains why nesting principle (e) permits an activity (and systemactivity) module to be further structured only into activity modules.

In summary, a specification designer combining the general feature of parent/children priority with the nesting principles characterized above may:

— At the "top level", structure the specification into a fixed configuration of independent, asynchronously running systems or (by making the specification module a system module) into a single synchronous system.

— Substructure each systemprocess module into submodules in such a way that their instances may run in parallel if they are not in ancestor/descendant conflict, i.e., the parallelism is synchronized by the supervising systemprocess.

— Decide, at each level of substructuring, that beginning from this level, instances of descending modules run in a nondeterministic way with preservation of a parent/children priority. This is accomplished by attributing the instance of the chosen level with the keyword **activity** (or **systemactivity** if it is the system level). In that case all descendant modules are also activity modules.

5.3 Specification behavior

5.3.1 Local situations

A *local situation* of a single module instance P is given by a pair (s_P, t_P) where s_P is one of the states of P and t_P is a transition of P. The transition t_P is said to be *offered* by P in s_P. The situation (s_P, null) denotes that no action is offered by P in s_P, which sometimes will be described by saying the "null transition" is offered.

A non-null transition offered in s_P must be a transition that is *fireable* in s_P. The set of *fireable transitions* in s_P is a subset of all enabled transitions in this state, selected on the basis of transition priorities and delays. Hence, strictly speaking, a transition t_P is said to be offered by P in s_P with respect to given delay values. The detailed definitions of enabled, fireable, and offered transitions are given in 9.6.3 and 9.6.5.

NOTE — The idea is that several transitions can be offered (nondeterminism), but determining which one is actually selected is an autonomous decision of the instance P. This property of autonomy is used below to establish precisely how a global situation changes after a transition's execution. It cannot be predicted, however, which next transition (among all those which are fireable) is going to be selected in the resulting situation. Nevertheless, each time, a module instance P makes the choice.

5.3.2 Global instantaneous description of a module instance

Consider the tree of module instances, each in one of its possible states, where the edges represent the parent/child relationship. This tree is defined in 9.5.3. If P is the root of such a tree, then the tree together with the states of its components is called a *global instantaneous description of* P (abbreviated gid_P). If the state of P is initial, then gid_P is called *initial*. (See 9.5.3 and 9.6.3.)

Example

Figures 1(b) and 1(c) give the trees of initial global instantaneous descriptions of an instance of the **specification** SP. Subtrees of these descriptions constitute the trees of global instantaneous descriptions of the specification components, e.g. in figure 1(c), subtrees rooted at the instance of systemprocess module A, the instance of systemactivity module B1, or the instance of activity module A1.

5.3.3 Transitions selected for execution

Consider an instance P of an attributed module (see Remark below) in one of P's global instantaneous descriptions gid_P. Assume that each module instance component of gid_P is in one of its local situations, i.e., that in its state (specified by gid_P) the module instance is offering a transition. Denote by AS(gid_P) a set of transitions *selected for execution* in gid_P; i.e., AS is the set of transitions selected from among those offered by the entire set of module instances, hierarchically structured in the tree of gid_P of which P is the root. There may be more than one AS set for a given gid_P due to the nondeterministic choice in case of activities (or system activities). The definition below characterizes all of them.

Remark: If P is an instance of an attributed module, then by the nesting principles of 5.2, P is a descendant of a system (or it is a system itself). Therefore the definitions below serve to characterize the synchronized parallelism within (i.e., among the descendant instances of) a systemprocess and nondeterminism within a systemactivity.

Definition of AS(gid_P):

(a) If the tree of gid_P is simply the single instance P, i.e., $gid_P = (P; s_P)$ and (s_P, t_P) is the local situation of P in gid_P, then AS(gid_P) = $\{t_P\}$ (the set $\{null\}$ is identified with the empty set).

(b) Let $P_1, P_2, \ldots, P_k$, $k \geq 1$, be children instances of P in the tree of gid_P, and let (s_P, t_P) be the local situation of P in gid_P.

(1) If $t_P \neq null$, then AS(gid_P) = $\{t_P\}$ (parent priority).

(2) If t_P = null and P is an activity or systemactivity (this implies that each P_i is an activity), then $AS(gid_P)$ equals one of the nonempty sets $AS(gid_{P_i})$, $i = 1, \ldots, k$, if such exists, and is empty otherwise. The choice is nondeterministic. (Notice that each $AS(gid_{P_i})$ is either empty or consists of exactly one transition since activities are substructured only into activities.)

(3) If t_P = null and P is a process (or systemprocess), then $AS(gid_P) = \bigcup_{i=1}^{k} AS(gid_{P_i})$.

NOTE — The above definition has the simple sense discussed in 5.2. If P is a system module instance, then $AS(gid_P)$ denotes the set of transitions that this system may currently (i.e., in gid_P) perform and will actually execute, in a synchronized and parallel way, from among those transitions which are chosen autonomously by every component instance of the system. Point (b)(1) expresses the parent/children priority principle; point (b)(2) shows nondeterministic execution among a hierarchy of activities, and point (b)(3) indicates that every transition offered by a process within a system, if not constrained by a ancestor/descendant priority conflict, will be selected and executed.

Example

In Figure 1(c) consider the tree of gid for the **systemprocess** which is an instance of the module A (in this example, declared names of modules are used as names of their instances).

(a) Assume:
null is offered (locally) by A
null is offered by A1
t11 is offered by A11
t12 is offered by A12
t2 is offered by A2.
In this case: $AS(gid_A)$ is either $\{t11, t2\}$ or $\{t12, t2\}$.

(b) Assume:
same as in (a) but t1 is offered by A1.
In this case: $AS(gid_A) = \{t1, t2\}$.

(c) Assume:
as in (b) but t is offered by A.
In this case: $AS(gid_A) = \{t\}$.

Remark: As observed earlier, for a given gid_P, there may exist more than one set of transitions selected for execution. This is due both to the nondeterministic choice in the case of activities (point (b)(2) of the above definition, illustrated by Example 3(a)), and to the local nondeterministic choice in selecting a transition by a module involved. Therefore, in the text that follows, $AS^*(gid_P)$ will denote the set of all possible sets of transitions selected for execution in a given gid_P, and $AS(gid_P)$ will denote an element in $AS^*(gid_P)$.

Note finally that each set AS (and in consequence the set AS*) may depend on the current delay values and not only on gid_P. As was indicated in 5.3.1, the transition to be offered by a module instance may depend on these delay values. The precise nature of this dependency is defined in 9.6.5.

5.3.4 Global behavior

Let SP denote an instance of the specification module (the main module in Estelle). The behavior of SP is defined operationally; i.e., for a given global instantaneous description of SP, the set of all possible next global instantaneous descriptions of SP is characterized. The computation starts with any of SP's initial gid; i.e., it is assumed that the specification module instance already exists in one of its initial states which, in turn, determines its initial gid. As noted above, the specification's initial state (therefore a corresponding gid) defines a fixed number of system module instances. These systems never cease to exist and the link structure above them in the hierarchy is fixed for the lifetime of the specification (if the specification's initial state does not contain a system module instance, then the specification behavior degenerates to this initial state).

Denote these system modules $S_1, \ldots, S_n$, where $n \geq 1$, and n may vary depending on the initial gid of SP.

Notice that, for every gid_{SP}, its part rooted at $S_i (i = 1, \ldots, n)$ is a global instantaneous description of S_i; denote this gid_{SP}/S_i. Similarly, let $AS(gid_{SP}/S_i)$ denote the set of transitions selected for execution from among those transitions offered by the entire subtree of module instances rooted at S_i. This tree is called *the system rooted at* S_i *in* gid_{SP}. Transitions once selected (i.e., permitted to run) eventually execute.

By a *global situation of* SP, we mean a tuple: $(gid_{SP}; A_1, \ldots, A_n)$, where each A_i is a set of transitions of the component instances of the system rooted at S_i.

A global situation of SP is said to be *initial* if and only if

gid_{SP} is initial, and
$A_i = \emptyset$, for $i = 1, \ldots, n$.

Each transition t of a component module instance may change internal objects of the instance (state) and, by its outputs, also change environments of others. Because t may contain nondeterministic instructions, it may transform a gid in more than one possible way. Given a gid_{SP} and a transition t, in this clause, we will use the notation $t(gid_{SP})$ to represent one of the possible resulting global instantaneous descriptions and the notation $t^*(gid_{SP})$ to represent the set of all possible resulting global instantaneous descriptions. Precise definitions of these are given in 9.5.3 and 9.5.4. Clearly, $t(gid_{SP}) \in t^*(gid_{SP})$.

The following property is assumed below:

if two transitions t_i and t_j are selected by S_i and S_j, respectively, and both are defined as partial functions for gid_{SP}, then t_i is also defined for any $t_j(gid_{SP})$ and similarly, t_j is defined for any $t_i(gid_{SP})$.

NOTE — For a given gid_P, there are only three ways the execution of a transition t_i may make another transition t_j undefined: either t_i changes values of some variables of the module instance, call it P_j, the transition t_j belongs to, or it releases the module instance P_j, or it detaches an interaction point of the module instance P_j. In each of these cases, t_i must be a transition of the parent instance of the module instance P_j in gid_{SP}. This is not possible by the parent/children priority principle of the definition of $AS(gid_{SP})$ from 5.3.3.

Intuitively speaking the above property says that systems rooted at P_i may really run independently despite the fact that they interchange messages. Due to the above property, this possible concurrent execution may be adequately expressed by interleaving.

Given a global situation, $sit = (gid_{SP}; A_1, \ldots, A_n)$, the set of next global situations is described as follows:

For every $i = 1, \ldots, n$,

(a) if $A_i = \emptyset$, then for every $AS(gid_{SP}/S_i) \in AS^*(gid_{SP}/S_i)$,

$(gid_{SP}; A_1, \ldots, AS(gid_{SP}/S_i), \ldots, A_n)$ is a next global situation of sit.

(b) if $A_i \neq \emptyset$, then for every $t \in A_i$,

$(t(gid_{SP}); A_1, \ldots, A_i \setminus \{t\}, \ldots, A_n)$ is a next global situation of sit.

NOTES

1 There are as many next global situations of the situation sit as there are possible choices of next transition t (and its result) in case (b), and different empty sets A_i in sit, for the case (a). In addition, in case (a), all possible choices of the set AS resulting from nondeterminism of each component process in the system rooted at S_i must be taken into account.

2 The above definition expresses the fact that no assumptions are made about the relative speed of execution of transitions. All possible interleavings of them must be considered. Notice, however, that once a transition is in the set A_i, it is considered to be executing and cannot be suspended as a result of another transition which completed its task earlier. In other words, interleaving is only a means of expressing concurrency without any implicit possibility of waiting an unspecified amount of time before actually executing.

3 Once a set of transitions $AS(gid_{SP}/S_i)$ selected by the system S_i for execution is actually executed, then a new set AS is generated in the new global instantaneous description "projected" onto the system. If this AS set is empty, then the system will try again in its next "system snapshot", the duration of which is unknown. This is expressed by (a). Alternatively, one may say that a system executes a "null" transition. Thus, the moment when a new set AS of actions is generated is solely determined by the system S_i. This means that each system runs independently from the others.

A sequence of global situations of SP, $sit_0, sit_1, \ldots, sit_j, \ldots$, is called a *computation* of SP if and only if sit_0 is initial, and for every $j > 0$, sit_j is one of the next global situations of sit_{j-1} as described by (a) or (b) above.

A global situation $sit_j (j > 0)$ in a computation is a *snapshot* of the system $S_i (i = 1, \ldots, n)$ if it is a next situation of sit_{j-1} described by (a) above (i.e. it is a situation in which a new AS set is chosen for the i-th system).

5.3.5 Concept of time in the model

Some Estelle transitions may contain a **delay-clause** (see 7.5.7). The intention of the **delay-clause** is to indicate that a transition's execution (if it is enabled in a state) should be delayed. Two times are associated with a delay: the minimum time the transition must be delayed and the maximum time it may be delayed. These are initially specified by two values of integer expressions occurring in the **delay-clause**. The dynamic change (non-increasing) of these time values with respect to the dynamic change of global situations, as described in the previous clause, relates the progress of time and computation.

The computational model for Estelle (as described in 5.3.3 and 5.3.4) is intentionally formulated in time-independent terms: one of the principal assumptions of this model is that nothing is known about the execution time of a transition in a module instance. Knowledge of execution speeds is considered implementation dependent. That is why all appropriate sequences of interleaved transitions have been considered as possible computations. In this perspective, no specific relationship between a time unit and execution speed of a module instance can be known or taken into account.

Consequently, delay values are assumed in this model to be dynamically changed by a "time process" which is independent of a specification. That is also the reason a timescale, optionally given in the specification, is treated as semantically irrelevant; it is merely an indication of the designer's intentions.

The only assumptions about time (i.e., about this external "time process" whose results are observed in terms of delay values during computations) that are taken into account in this semantic model are that

Meta-symbol	Meaning
=	shall be defined to be
|	alternatively
.	end of definition
[x]	0 or 1 instance of x
{x}	0 or more instances of x
+{x}	1 or more instances of x
(x | y)	grouping: either of x or y
x⧢y	xy | yx
a_1⧢···⧢a_n	all possible strings consisting of all the elements concatenated in an arbitrary order
"xyz"	the terminal symbol xyz
meta-identifier	a nonterminal symbol

Table 1 — Metalanguage Symbols

(a) time progresses as the computation does, and

(b) this progression is uniform with respect to delay values of transitions involved.

Point (b) means that given two enabled and delayed transitions in a computation situation, if they remain enabled in the following situation, then their delay values would decrease (if at all) by the same amount. Any other constraints are considered implementation dependent. Any constraint, however, which does not contradict the above assumption is admissible, e.g., for simulation purposes.

The technical and precise formulation of the approach sketched above can be found in 9.6.4 and 9.6.5.

6 Definitional Conventions

6.1 Syntax definitions

The metalanguage used in this International Standard to specify the syntax of constructs is based on Backus-Naur Form. The notation has been modified from the original to permit greater convenience of description. Table 1 lists the meanings of the various metasymbols.

A meta-identifier shall be a sequence of letters and hyphens beginning with a letter. A sequence of terminal and nonterminal symbols in a production implies the concatenation of the text that they ultimately represent. Within clauses 7 and 8 and Annexes A and C and their subclauses this concatenation shall be direct; no characters shall intervene. In all other parts of this International Standard the concatenation shall be in accordance with the grammar rules set out in clauses 7, 8 and Annex C.

Syntax rules are defined in clauses 7, 8 and Annex C; the relationship between the parts is as follows. Annex C defines a subset of ISO Pascal [ISO/IEC 7185:1990] used by Estelle. Clause 8 defines extensions to the subset defined in annex C and summarizes the restrictions imposed by the subset. Clause 7 defines those grammar elements that are unique to Estelle. By convention, in clauses 7 and 8, nonterminal symbols written entirely in upper case refer to the corresponding lower case nonterminal symbol in the grammar rules found in Annex C (e.g., **IDENTIFIER**).

Note that some nonterminals found in Annex C are extended by clause 8. For example, an applied occurrence of **COMPONENT-VARIABLE** in clause 7 refers to the nonterminal symbol "component-variable" as defined in Annex C and as extended in clause 8.

NOTE — The start symbol of the grammar is the nonterminal symbol **specification** (see 7.2.1).

A complete listing of the syntactic elements is presented in Annex A.

Use of the words "of", "in", "containing", and "closest-containing" when expressing a relationship between terminal or nonterminal shall have the following meanings:

the x of a y: refers to the x occurring directly in a production defining y

the x in a y: is synonymous with "the x of a y"

a y containing an x: refers to any y from which an x is directly or indirectly derived

the y closest-containing an x: that y which contains an x but which does not contain another y containing that x.

These syntactic conventions are used in clauses 7, 8 and Annex C to specify certain syntactic requirements and also the contexts within which certain semantic specifications apply.

By convention, when terminal and nonterminal symbols appear in prose descriptions of constraints and informal semantics, these symbols are in **bold** font.

6.2 Semantic notations

The following is a summary of the most important notations used in clauses 5 and 9.

Sets:

$\emptyset$ — empty set

$\in$ — set element

$\cup$ — set union

$\cap$ — set intersection

$\bigcup_{x \in A} P(x)$
— union of all sets P(x) defined over the index set A

$\setminus$ — set subtraction

$\times$ — set cartesian product

POWER(A)
— the set of all subsets of the set A

A^* — the set of finite sequences (lists) of elements of the set A

Lists:

nil — the empty list

head
— head(L) = a if L = aL′, i.e., **head** is the function defined for nonempty lists that returns the first element of the list L

tail — $\mathrm{tail}(L) = L'$ if $L = aL'$, i.e., **tail** is the function defined for nonempty lists that returns the list which remains after removing the first element from L

append
— $\mathrm{append}(L', L) = LL'$, i.e., **append** is the function that, for two arbitrary lists L' and L, returns the list which is the concatenation of L with L'

Functions:

$f : A \rightarrow B$
— f is a total function from the set A into the set B

$f : A \rightsquigarrow B$
— f is a partial function from the set A into set B (note that every total function is also partial)

dom(f)
— the set of all elements in the set A for which the (partial) function $f : A \rightsquigarrow B$ is defined (the domain of f)

$f : A \rightsquigarrow B \rightsquigarrow C$ means $f : A \rightsquigarrow (B \rightsquigarrow C)$ (similarly for $\rightarrow$)

$f(a)(b) = (f(a))(b)$
— see above

f[g] — the covering operator which for any two partial functions $f : A \rightsquigarrow B$ and $g : C \rightsquigarrow D$ returns the function $f[g] : A \cup C \rightsquigarrow B \cup D$ with:

$$\begin{aligned} \mathrm{dom}(f[g]) &= \mathrm{dom}(f) \cup \mathrm{dom}(g), \text{ and} \\ f[g](a) &= \begin{cases} g(a) & \text{if } a \in \mathrm{dom}(g) \\ f(a) & \text{if } a \notin \mathrm{dom}(g) \end{cases} \end{aligned}$$

$f(a_1/v_1, \ldots, a_n/v_n)$
— simultaneous substitution, i.e., an operator that, for a given function $f : A \rightsquigarrow B$, $a_1, \ldots, a_n \in \mathrm{dom}(f)$ and $v_1, \ldots, v_n \in B$, returns the function of the same domain and identically defined as f for all $a \neq a_i$ $(i = 1, \ldots, n)$, and assuming the value v_i for an argument a_i

Meta-expressions:

The metalanguage used for semantic definitions is a mixture of conventional mathematical notation and the following programming-like constructs, which are assumed to have known meanings:

Assignment: meta-var := meta-exp

Conditional: **if** meta-exp **then** meta-stm_1 **else** meta-stm_2

Forall statement: **forall** $x \in$ Set **do** meta-stm

Block: **let** local meta-definitions and/or assumptions **in** meta-stm

Compound meta-statement: $(\text{meta-stm}_1; \text{meta-stm}_2; \ldots; \text{meta-stm}_n)$

Remark 1: All the above constructs are used in a relatively functional style despite the presence of assignments. They serve, in most cases, to define a result of state transformation.

Remark 2: Any state is a complex object: $S = (s.ie, s.Loc, \ldots, s.out)$. An assignment to a state component, e.g., $s.Loc := f(s.Loc, \ldots)$ describes a state transformation which for a given state s returns a new state that is identical to s except that s.Loc component has changed in the way given by the function f.

Remark 3: The meta-stm in a forall statement is executed for each element in the Set of this statement. The result of a forall meta-statement never depends in this International Standard on the order of execution except when it is used to explain the **all-statement** of Estelle.

Remark 4: Local definitions and assumptions in a block (those between "let" and "in") are valid only within the closest following (compound) meta-statement.

Special conventions:

INST(M, B, E)
— the set of instances of the module given by a module-header-definition M and a module-body-definition B in a context environment E

INST(M, E)
— the set of instances of any module given by a module-header-definition M in a context environment E

[Estelle-statement]$_P$
— interpretation of an Estelle statement in a module instance P (as a partial function from the instance's states into sets of instance's states); i.e., [ ]$_P$ denotes the semantics of statements.

val$_P$(Estelle-expression)
— interpretation of an Estelle expression in a module instance P (as a partial function from the instance's states into the set of values); i.e., val$_P$ denotes the semantics of expressions.

7 Language elements

7.1 Introduction

This clause defines the elements of the language, giving the syntax with BNF grammar rules, the context dependent static constraints and scope rules, and the informal semantics with references to formal semantics.

7.1.1 Character set

The characters appearing in Estelle specifications shall be those defined in ISO/IEC 646:1991.

NOTE — As noted in Annex C, the representation of any letter (upper-case or lower-case, differences of font, etc.) occurring anywhere outside of a character-string (see Annex C, 6.1.7) shall be insignificant in that occurrence to the meaning of the specification.

7.1.2 Estelle scope rules

7.1.2.1

Each **IDENTIFIER** or **LABEL** contained by the **body-definition** of the specification shall have a defining-point.

7.1.2.2

Each defining-point shall have a region that is a part of the specification text, and a scope that is a part or all of that region.

7.1.2.3

The region of each defining-point is defined in clauses 7, and 8, and in Annex C.

Remark: In the clauses above referencing the modified Pascal Standard (Annex C, clause 6), all references to "**block**" shall be replaced by "**block** or **transition-block** or **body-definition**".

7.1.2.4

The scope of each defining-point shall be its region (including all regions enclosed by that region) subject to the constraints in clauses 7 and 8, and in Annex C. (Note especially 7.1.2.6).

7.1.2.5

When an **IDENTIFIER** or **LABEL** has a defining-point for region A and another **IDENTIFIER** or **LABEL** having the same spelling has a defining-point for some region B enclosed by A, then region B and all regions enclosed by B shall be excluded from the scope of the defining-point for region A.

7.1.2.6

The region that is the **FIELD-SPECIFIER** of a **FIELD-DESIGNATOR** shall be excluded from the enclosing scopes.

The region that is the **role-identifier** of an **interaction-point-type** shall be excluded from the enclosing scopes.

The region that is the **external-ip** of a **child-external-ip** shall be excluded from the enclosing scopes.

The region that is the **exported-variable-identifier** of an **exported-variable** shall be excluded from the enclosing scopes.

The region that is the **interaction-identifier** of an **interaction-reference** shall be excluded from the enclosing scopes.

The region that is the **interaction-argument-list** of a transition shall be excluded from the enclosing scopes.

7.1.2.7

When an **IDENTIFIER** or **LABEL** has a defining-point for a region, another **IDENTIFIER** or **LABEL** with the same spelling shall not have a defining-point for that region.

7.1.2.8

Within the scope of a defining-point of an **IDENTIFIER** or **LABEL**, each occurrence of an **IDENTIFIER** or **LABEL** having the same spelling as the **IDENTIFIER** or **LABEL** of the defining-point shall be designated an *applied occurrence* of the **IDENTIFIER** or **LABEL** of the defining-point, except for an occurrence that constituted the defining-point of that **IDENTIFIER** or **LABEL**; such an occurrence shall be designated a *defining occurrence*. No occurrence outside that scope shall be an applied occurrence.

NOTE — Within the scope of a defining-point of an **IDENTIFIER** or **LABEL**, there are no applied occurrences of an **IDENTIFIER** or **LABEL** that cannot be distinguished from it and have a defining-point for a region enclosing that scope.

7.1.2.9

The defining-point of an **IDENTIFIER** or **LABEL** shall precede all applied occurrences of that **IDENTIFIER** or **LABEL** contained by the **body-definition** of the specification with one exception, namely that an **IDENTIFIER** may have an applied occurrence in the **TYPE-IDENTIFIER** of the **domain-type** of any **new-pointer-types** contained by the **type-definition-part** that contains the defining-point of the **TYPE-IDENTIFIER**.

7.1.2.10

Identifiers that denote required constants, types, procedures and functions shall be used as if their defining-points have a region enclosing the specification (see Annex C, C.6.1.3, C.6.3, C.6.4.1 and C.6.6.4.1).

7.1.2.11

Whatever an **IDENTIFIER** or **LABEL** denotes at its defining-point shall be denoted at all applied occurrences of that **IDENTIFIER** or **LABEL**.

NOTE — Within syntax definitions, an applied occurrence of an **IDENTIFIER** is qualified, e.g. **TYPE-IDENTIFIER**, whereas a use that constitutes a defining-point is not qualified.

7.2 Structure of a specification

7.2.1 Syntax

```
specification = "specification" IDENTIFIER [ system-class ] ";"
                [ default-options ]
                [ time-options ]
                body-definition
                "end" "." .


system-class = "systemprocess" | "systemactivity" .
```

```
default-options  =  "default" queue-discipline ";" .

queue-discipline  =  "common" "queue" | "individual" "queue" .

time-options  =  "timescale" IDENTIFIER ";" .

body-definition  =  declaration-part
                    initialization-part
                    transition-declaration-part .
```

7.2.2 Constraints

The **IDENTIFIER** of the **specification** shall be the **specification** name which shall have no significance within the **specification**.

If a **specification** is declared with the **systemactivity** keyword as its **system-class**, then it is said to have the systemactivity attribute and all enclosed modules (if any) must be declared with the activity attribute. The **specification** shall have a **system-class** if its **body-definition** closest contains a non-empty **transition-declaration-part**.

The **default-options** define the queuing discipline which is to be used for any interaction point declaration in the specification which has no explicit **queue-discipline** specified. A default **queue-discipline** shall be provided if any interaction point within the entire specification is declared without a **queue-discipline**.

If specified, the **timescale** option defines the default units for all **delay** clauses specified for transitions contained within the specification. See 7.5.7.

The set of allowable **IDENTIFIERs** for the **time-options** shall include the identifiers **hours**, **minutes**, **seconds**, **milliseconds**, and **microseconds**.

7.2.3 Interpretation rules

A specification module with a default option included is semantically equivalent to the same specification without this option where, for any **interaction-point-declaration** without a **queue-discipline**, the default option above is added.

The **time-options** clause shall have no semantic significance.

The interpretation of a specification such as:

```
specification identifier [ system-class ];
    body-definition
end.
```

shall be the same as the interpretation of the following module (see 7.3.6, 7.3.7, and clause 9); i.e., instances of such a specification are defined as instances of the module below:

```
module header-identifier [ system-class ];
end;
body identifier for header-identifier;
      body-definition
end;
```

where the header-identifier in the **module-header-definition** and **module-body-definition** are the same identifier, and the body identifier and the specification identifier are the same, and the **body-definition** within the specification is identical with the one in the **module-body-definition**, and the keyword **systemactivity** or **systemprocess** appears in the specification if, and only if, it appears in the module header.

7.2.4 Informal semantics

Initially, it is assumed that a specification instance exists in one of the initial states that results from executing a transition from its **initialization-part** (7.5.10) if the **initialization-part** is non-empty. If the **initialization-part** is empty, then the instance exists in a *preinitial* state, as defined in 9.4.5.2.

The global behavior described in 5.3 applies only to collections of systems of modules defined by a specification.

7.3 Declaration part

The clauses of the **declaration-part** are given below. They may occur in any order and each, except for the **state-definition-part**, may occur more than once.

7.3.1 Syntax

```
declaration-part = { declarations } .


declarations = CONSTANT-DEFINITION-PART
             | TYPE-DEFINITION-PART
             | channel-definition
             | module-header-definition
             | module-body-definition
             | interaction-point-declaration-part
             | module-variable-declaration-part
             | VARIABLE-DECLARATION-PART
             | state-definition-part
             | state-set-definition-part
             | PROCEDURE-AND-FUNCTION-DECLARATION-PART .
```

7.3.2 Constraints

When an **IDENTIFIER** has a defining-point as a **variable-identifier** for region A and there is a region B enclosed by A which is a **body-definition**, then region B and all regions enclosed by B shall be excluded from the scope of the defining-point for region A.

When an **IDENTIFIER** has a defining-point as a **procedure-identifier** or **function-identifier** for region A, and there is a **procedure-block** or **function-block** following the **procedure-heading** or **function-heading** which is a **block** (not **primitive**), and there is a region B enclosed by A which is a **body-definition**, then region B and all regions enclosed by B shall be excluded from the scope of the defining-point for region A.

There shall be at most one **state-definition-part** in each **body-definition**.

7.3.3 Informal semantics

A **declaration-part** of a **body-definition** establishes sets of permissible values for each internal object of each instance of a module with this **body-definition** (i.e., the set of states of an instance; see 5.1 and 9.4) and the external context environment in which the instances of the children modules are defined (see 9.3).

The **CONSTANT-DEFINITION**, **TYPE-DEFINITION**, **VARIABLE-DECLARATION**, and **PROCEDURE-AND-FUNCTION-DECLARATION** parts are given in clause 8 and Annex C.

7.3.4 Channel definition

7.3.4.1 Syntax

```
channel-definition = channel-heading channel-block .

channel-heading = "channel" IDENTIFIER "(" role-list ")" ";" .

channel-identifier = IDENTIFIER .

role-list = IDENTIFIER "," IDENTIFIER .

role-identifier = IDENTIFIER .

channel-block = +{ interaction-group } .
```

```
interaction-group =  "by" role-identifier [ "," role-identifier ] ":"
                     +{ interaction-definition } .


interaction-definition =  IDENTIFIER
                          [ "(" VALUE-PARAMETER-SPECIFICATION
                          { ";" VALUE-PARAMETER-SPECIFICATION } ")"] ";" .


interaction-argument-identifier =  IDENTIFIER .


interaction-identifier =  IDENTIFIER .
```

7.3.4.2 Constraints

The occurrence of an **IDENTIFIER** in the **channel-heading** of a **channel-definition** shall constitute its defining-point as a **channel-identifier** for the region that is the **body-definition** closest-containing the **channel-definition**.

The occurrence of an **IDENTIFIER** in the **role-list** of a **channel-heading** of a **channel-definition** shall constitute its defining-point as a **role-identifier** for the region that is the **channel-definition**.

Each **role-identifier** referenced in an **interaction-group** shall have been previously specified in the **role-list** of the **channel-heading** of the **channel-definition** that closest-contains the **interaction-group**.

The occurrence of an **IDENTIFIER** in the **interaction-definition** of an **interaction-group** shall constitute its defining-point as a **interaction-identifier** for the region that is the **channel-definition**.

The occurrence of an **IDENTIFIER** in the **value-parameter-specification** of an **interaction-definition** shall constitute its defining-point as a **interaction-argument-identifier** for the region that is the **interaction-definition**.

A type shall be designated as *pointer-containing* if it is a **pointer-type** or it is a **structured-type** possessing a **component-type** that is pointer-containing.

A **VALUE-PARAMETER-SPECIFICATION** of an **interaction-definition** of a **channel-definition** shall not contain a **TYPE-IDENTIFIER** denoting a pointer-containing type.

7.3.4.3 Interpretation rules

Each **interaction-definition** within an **interaction-group** is said to be *associated* with the **role-identifier**(s) of the group.

Any **channel-definition** is semantically equivalent to a channel definition with exactly three **interaction-groups**: one for the first **role-identifier**, one for the second, and one for both **role-identifiers** of the **role-list**.

The first **interaction-group** includes all **interaction-definitions** associated with the first **role-identifier** and not associated with the second.

The second **interaction-group** includes all **interaction-definitions** associated with the second **role-identifier** and not associated with the first.

The third **interaction-group** includes all **interaction-definitions** associated with both **role-identifiers**.

Example:

```
channel H(R1,R2);                              channel H(R1,R2);
    by R1 : m1;          is equivalent to          by R1 : m1;
    by R2 : m2;                                    by R2 : m2; m4;
    by R1, R2 : m3;                                by R1, R2 : m3; m5;
    by R2 : m4;
    by R1, R2 : m5;
```

7.3.4.4 Informal semantics

For a given interaction point, a module assumes a role declared by a **role-identifier** contained in the **role-list** of a **channel-heading**. A module may send an interaction associated with its assumed role. A module that assumes the opposite role may receive interactions associated with the first role. (See also module interconnection, 7.6.3 to 7.6.7.)

Remark: Each of the two **role-identifiers** in the **role-list** is said to be the *opposite* of the other.

Each **interaction-definition** in an **interaction-group** is said to be associated with each **role-identifier** of this **interaction-group**.

7.3.5 Interaction points

NOTE — An interaction point has three attributes:

— the channel identifier referenced;

— a role identifier specifying which interactions the module may send and which it may receive;

— the queuing discipline to be used for interactions received through the interaction point.

Interaction points may be declared in two forms: external (7.3.6.1) or internal (7.3.8). Like interaction points may be grouped into arrays.

7.3.5.1 Interaction point declaration part

7.3.5.1.1 Syntax

interaction-point-declaration-part = "ip" +{ interaction-point-declaration ";" } .

```
interaction-point-declaration =  IDENTIFIER-LIST ":" interaction-point-type
                               | IDENTIFIER-LIST ":" "array" "[" index-type-list "]"
                                                      "of" interaction-point-type  .
```

```
interaction-point-identifier =  IDENTIFIER  .
```

```
interaction-point-type =  channel-identifier "(" role-identifier ")" [ queue-discipline ]  .
```

```
index-type-list =  INDEX-TYPE { "," INDEX-TYPE }  .
```

7.3.5.1.2 Constraints

NOTE — Interaction point identifiers may be referenced only in the binding operations **connect** and **attach** (7.6.3 and 7.6.4), the unbinding operations **disconnect** and **detach** (7.6.5 and 7.6.6), the **when** clause of a **transition** (7.5.6), and in the **output** statement (7.6.8).

If a **queue-discipline** is not given in an **interaction-point-declaration**, then the default **queue-discipline** of the specification shall have been specified, and the default **queue-discipline** shall be used as the **queue-discipline** of the interaction point.

An **INDEX-TYPE** of an **index-type-list** of an **interaction-point-declaration** shall be a finite **ORDINAL-TYPE**.

The occurrence of a **channel-identifier** in an **interaction-point-type** shall constitute the defining-point of the **role-identifiers** associated with that **channel-identifier** for the region that is the **interaction-point-type**.

NOTE — The **role-identifier** of the **interaction-point-type** must be one associated with the channel-identifier of the **interaction-point-type**.

7.3.5.1.3 Informal semantics

An **interaction-point-identifier** identifies a single interaction point or an array of interaction points.

An interaction point is an abstract, bidirectional interface through which a module may send and receive interactions.

An **interaction-point-type** references a **channel-identifier** and a **role-identifier** associated with that **channel-identifier**. Any interaction associated with that role may be sent through an interaction point of this **interaction-point-type**. Any interaction associated with the opposite role may be received through an interaction point of this **interaction-point-type**.

The **queue-discipline** of an **interaction-point-type** determines whether the queue assigned to any interaction point of that type within a module instance is shared (**common**) or is not shared (**individual**) with other interaction points of that module instance (see 9.4.3).

7.3.5.2 External interaction points

External interaction points of a module are declared in the **interaction-point-declaration** of a **module-header-definition**.

7.3.5.2.1 Constraints

The occurrence of an **IDENTIFIER** in the **interaction-point-declaration** of a **module-header-definition** shall constitute its defining-point as a **interaction-point-identifier** for the region that is the **module-header-definition** and shall associate the **interaction-point-identifier** with a distinct component of the **module-header-type**.

7.3.5.2.2 Informal semantics

External interaction points may be bound using connect and attach operations (see 7.6.3 and 7.6.4) and unbound using disconnect and detach operations (see 7.6.5 and 7.6.6).

7.3.6 Module header

7.3.6.1 Syntax

```
module-header-definition =  "module" IDENTIFIER [class] [ "(" parameter-list ")" ] ";"
                            [ "ip" +{ interaction-point-declaration ";" } ]
                            [ "export" +{ exported-variable-declaration ";" } ]
                            "end" ";" .


header-identifier =  IDENTIFIER .


class =  "systemprocess" | "systemactivity" | "process" | "activity" .


parameter-list =  VALUE-PARAMETER-SPECIFICATION
                  { ";" VALUE-PARAMETER-SPECIFICATION } .


exported-variable-declaration =  VARIABLE-DECLARATION .
```

7.3.6.2 Constraints

The occurrence of an **IDENTIFIER** in the **module-header-definition** shall constitute its defining-point as a **header-identifier** for the region that is the **body-definition** closest-containing the **module-header-definition**. When an

IDENTIFIER has a defining-point as a **header-identifier** for region A and there is a region B enclosed by A which is a **body-definition**, a **PROCEDURE-BLOCK**, or a **FUNCTION-BLOCK**, then region B and all regions enclosed by B shall be excluded from the scope of the defining-point for region A.

The occurrence of an **IDENTIFIER** in the **VALUE-PARAMETER-SPECIFICATION** of a **parameter-list** of a **module-header-definition** shall constitute its defining-point as a **module-parameter-identifier** for the region that is the **module-header-definition** and its defining-point as the associated **variable-identifier** for any region that is the **body-definition** of a **module-body-definition** having an applied occurrence of the **header-identifier** of the **module-header-definition**.

The occurrence of an **IDENTIFIER** in the **interaction-point-declaration** of a **module-header-definition** shall constitute its defining-point as a **interaction-point-identifier** for the region that is the **module-header-definition** and shall *associate* the **interaction-point-identifier** with the **header-identifier**.

NOTE — Module parameters are passed by value (see 9.6.6.1).

A **VALUE-PARAMETER-SPECIFICATION** of a **parameter-list** of a **module-header-definition** shall not contain a **TYPE-IDENTIFIER** denoting a pointer-containing type.

The optional keyword **systemactivity** or **systemprocess** shall be used in a **module-header-definition** if the **transition-declaration-part** of at least one associated **body-definition** is non-empty and no enclosing module is attributed with the keyword **systemactivity** or **systemprocess**.

The optional keyword **systemactivity** or **systemprocess** shall not be used if an enclosing (i.e., ancestor) module is attributed with the keyword **systemactivity** or **systemprocess**.

The optional keyword **activity** or **process** shall be in each enclosed (i.e., descendent) **module-header-definition** of a module that used a keyword **systemprocess** or **systemactivity**.

The optional keyword **activity** or **process** shall not be used in modules enclosing a module attributed with the keyword **systemprocess** or **systemactivity**.

A **module-body-definition** associated with a **module-header-definition** with a keyword **systemactivity** or **activity** shall not enclose a **module-header-definition** with a keyword **process**.

Remark: From the above constraints one may derive that:

— each active module shall be attributed,

— each ancestor of a module attributed as **systemprocess** or **systemactivity** shall be inactive and unattributed,

— each descendant of a module attributed as **systemprocess** or **process** shall be attributed as **process** or **activity**, and

— each descendant of a module attributed as **systemactivity** or **activity** shall be attributed **activity**.

7.3.6.3 Informal semantics

The module header defines defines the external visibility of a module in terms of its interaction points and exported variables (see 5.1).

Exported variables are variables belonging to any instance of the specified module which can be accessed by its parent. The rules for accessing these variables are given in 7.4.3.

The module header also defines parameters which are passed to an instance of the module when the instance is created (see 7.6.1 and 9.6.6.1). Parameters may be referenced in the body of the corresponding module. Since module parameters are passed by value, each module instance receives its own copy of the parameters with their actual values evaluated at the time a module instance is initialized.

Remark: The interaction points declared in the **module-header-definition** are called *external interaction points*.

7.3.7 Module body definition

The **module-body-definition** shall *associate* a **header-identifier** with a module **body-identifier** and its body.

7.3.7.1 Syntax

```
module-body-definition =  "body" IDENTIFIER "for" header-identifier ";"
                          ( body-definition "end" ";"  | "external" ";" ) .

body-identifier =  IDENTIFIER .
```

7.3.7.2 Constraints

The occurrence of an **IDENTIFIER** in the **module-body-definition** shall constitute its defining-point as a **body-identifier** for the region that is the **body-definition** closest-containing the **module-body-definition**. When an **IDENTIFIER** has a defining-point as a **body-identifier** for region A and there is a region B enclosed by A which is a **body-definition**, a **PROCEDURE-BLOCK**, or a **FUNCTION-BLOCK**, then region B and all regions enclosed by B shall be excluded from the scope of the defining-point for region A.

The occurrence of a **header-identifier** in a **module-body-definition** shall constitute a defining-point of each **interaction-point-identifier** associated with that **header-identifier** for the region that is the **module-body-definition**. When an **IDENTIFIER** has a defining-point as an **interaction-point-identifier** for region A and there is a region B enclosed by A which is a **body-definition**, a **PROCEDURE-BLOCK**, or a **FUNCTION-BLOCK**, then region B and all regions enclosed by B shall be excluded from the scope of the defining-point of region A.

The occurrence of a **header-identifier** in a **module-body-definition** shall constitute a defining-point of each **exported-variable-identifier** associated with that **header-identifier** for the region that is the **module-body-definition**. When an **IDENTIFIER** has a defining-point as an **exported-variable-identifier** for region A and there is a region B enclosed by A which is a **body-definition**, then region B and all regions enclosed by B shall be excluded from the scope of the defining-point of region A.

NOTE — From the scope rules for **module-header-identifiers** it follows that a **module-header-definition** and any associated **module-body-definition** must be immediately contained in the same **declaration-part** of a given module.

7.3.7.3 Informal semantics

Each **module-body-definition** and the associated **module-header-definition** constitute a single module-definition (see 5.1 and 9.1). If a **module-body-definition** contains the keyword **external**, then the specification in which the **module-body-definition** occurs is not complete, and it denotes a collection of specifications: one for each specific **body-definition** (followed by the keyword **end**) substituted in place of the keyword **external**.

There may be more than one **module-body-definition** associated with a given **module-header-definition**. Each **module-body-definition** defines one possible internal behavior of a module, where the external visibility is given by the **module-header-definition**. A body for a module is selected in an **init** statement when a module instance is created.

7.3.8 Internal interaction points

Internal interaction points shall be specified in an **interaction-point-declaration** within the **declaration-part** of a module **body-definition**.

7.3.8.1 Constraints

The occurrence of an **IDENTIFIER** in the **interaction-point-declaration** of an **interaction-point-declaration-part** shall constitute its defining-point as an **interaction-point-identifier** for the region that is the **body-definition** closest-containing the **interaction-point-declaration-part**. When an **IDENTIFIER** has a defining-point as a **interaction-point-identifier** for region A and there is a region B enclosed by A which is a **body-definition**, a **PROCEDURE-BLOCK**, or a **FUNCTION-BLOCK**, then region B and all regions enclosed by B shall be excluded from the scope of the defining-point for region A.

7.3.8.2 Informal semantics

Internal interaction points may be used to exchange interactions within a module instance or to exchange interactions with a child module instance. Internal interaction points may be bound using connect operations (see 7.6.3) and unbound using disconnect operations (see 7.6.5).

7.3.9 Module variable declaration part

Module variables identify instances of modules. Multiple instances of a single module may exist, so module variables serve to distinguish between instances of modules and form part of references to exported variables and interaction points of children modules.

7.3.9.1 Syntax

```
module-variable-declaration-part = "modvar" +{ module-variable-declaration ";" } .
```

```
module-variable-declaration =  IDENTIFIER-LIST ":" header-identifier
                              | IDENTIFIER-LIST ":" "array" "[" index-type-list "]"
                                                    "of" header-identifier .
```

7.3.9.2 Constraints

The occurrence of an **IDENTIFIER** in the **module-variable-declaration-part** shall constitute its defining-point as a **module-variable-identifier** for the region that is the **body-definition** closest containing the **module-variable-declaration-part**. When an **IDENTIFIER** has a defining-point as a **module-variable-identifier** for region A and there is a region B enclosed by A which is a **body-definition**, a **PROCEDURE-BLOCK**, or a **FUNCTION-BLOCK**, then region B and all regions enclosed by B shall be excluded from the scope of the defining-point for region A.

7.3.9.3 Informal semantics

Module variables provide unique identification of an instance of a module. The initial value of a **module-variable** is undefined. A value is assigned in an init statement. A release or terminate statement causes the value to be undefined.

Module variables are used by a parent module to qualify the reference to exported variables of children (7.4.3) and to qualify the references to interaction points of children when binding or unbinding pairs of interaction points (7.6.3 to 7.6.7).

7.3.10 State definition part

If the control state space of the EFSM (Extended Finite State Machine) underlying the module is nonempty, it shall be specified by enumeration. All values of the control state of the extended finite state machine shall be enumerated in a state definition part.

7.3.10.1 Syntax

```
state-definition-part =  "state" IDENTIFIER-LIST ";" .


state-identifier =  IDENTIFIER .
```

7.3.10.2 Constraints

The occurrence of an **IDENTIFIER** in the **state-definition-part** shall constitute its defining-point as a **state-identifier** for the region that is the **body-definition** closest containing the **state-definition-part**. When an **IDEN-**

TIFIER has a defining-point as a **state-identifier** for region A and there is a region B enclosed by A which is a **body-definition**, a **PROCEDURE-BLOCK**, or a **FUNCTION-BLOCK**, then region B and all regions enclosed by B shall be excluded from the scope of the defining-point for region A.

7.3.10.3 Informal semantics

The **state-definition-part** enumerates the possible values of the control state variable.

7.3.11 State set definition part

7.3.11.1 Syntax

```
state-set-definition-part = "stateset" +{ state-set-definition ";"} .

state-set-definition = IDENTIFIER "=" state-set-constant .

state-set-identifier = IDENTIFIER .

state-set-constant = "["state-identifier { "," state-identifier } "]" .
```

7.3.11.2 Constraints

The occurrence of an **IDENTIFIER** in the **state-set-definition** of a **state-set-definition-part** shall constitute its defining-point as a **state-set-identifier** for the region that is the **body-definition** closest-containing the **state-set-definition-part**. When an **IDENTIFIER** has a defining-point as a **state-set-identifier** for region A and there is a region B enclosed by A which is a **body-definition**, a **PROCEDURE-BLOCK**, or a **FUNCTION-BLOCK**, then region B and all regions enclosed by B shall be excluded from the scope of the defining-point for region A.

NOTE — State sets may be referenced only in a **from-clause** of a transition.

7.3.11.3 Informal semantics

State sets provide a means of referring to several states at once. A state set is simply a compact notation for the list of its elements; a **state-set-identifier** stands for the list of its elements.

7.4 References to Estelle objects

7.4.1 Module variable reference

A **module-variable** is a variable which identifies an instance of a module.

7.4.1.1 Syntax

```
module-variable =  module-variable-identifier
                 | module-variable-identifier "[" INDEX-EXPRESSION
                                              { "," INDEX-EXPRESSION } "]" .
```

```
module-variable-identifier =  IDENTIFIER .
```

7.4.1.2 Constraints

The **INDEX-EXPRESSION**(s) of a **module-variable** shall be compatible and in corresponding order with the **INDEX-TYPE**(s) contained in the corresponding **module-variable-declaration**.

The **INDEX-TYPE** of an **index-type-list** of a **module-variable-declaration** shall be a finite ordinal type.

7.4.1.3 Informal semantics

A **module-variable-declaration** specifies the class of module instances the **module-variable** may assume as values (see 9.4.1 and 9.4.5 as well as 7.6.1).

NOTE — Module variables are used by a parent module to qualify the reference to exported variables of children (7.4.3) and to qualify the references to interaction points of children when binding or unbinding pairs of interaction points (7.6.3 to 7.6.7).

7.4.2 Interaction point reference

7.4.2.1 Syntax

```
connect-ip =  child-external-ip | internal-ip .
```

```
child-external-ip =  module-variable "." external-ip .
```

```
external-ip =  interaction-point-reference .
```

```
internal-ip  =  interaction-point-reference  .


interaction-point-reference  =  interaction-point-identifier
                                [ "[" INDEX-EXPRESSION { "," INDEX-EXPRESSION } "]" ]  .
```

7.4.2.2 Constraints

The occurrence of a **module-variable** in an **child-external-ip** shall constitute a defining-point for each **interaction-point-identifier** associated with components of the **module-header-definition** possessed by the **module-variable**.

The **interaction-point-identifier** of an **interaction-point-reference** of an **external-ip** shall have a defining-point in an **interaction-point-declaration** of a **module-header-definition**.

The **interaction-point-identifier** of an **interaction-point-reference** of an **internal-ip** shall have a defining-point in an **interaction-point-declaration** of an **interaction-point-declaration-part** of a **body-definition**.

Any **INDEX-EXPRESSION** of an **interaction-point-reference** shall be compatible and in corresponding order with the **INDEX-TYPE**(s) contained in the corresponding **interaction-point-declaration**.

7.4.2.3 Informal semantics

An **interaction-point-reference** identifies an interaction point, either external or internal, of a module instance, or an external interaction point of one of its child instances (see 9.6.1).

7.4.3 Exported variable reference

7.4.3.1 Syntax

```
exported-variable  =  module-variable "." exported-variable-identifier  .


exported-variable-identifier  =  IDENTIFIER  .
```

7.4.3.2 Constraints

The occurrence of an **IDENTIFIER** in the **VARIABLE-DECLARATION** of an **exported-variable-list** of a **module-header-definition** shall constitute its defining-point as a **exported-variable-identifier** for the region that is the **module-header-definition**, and shall associate the **exported-variable-identifier** with the **header-identifier**.

Exported variable types shall not be pointer-containing (see 7.3.4.2).

7.4.3.3 Informal semantics

The exported variable refers to the denoted variable of the specified module.

Example:

```
modvar
    X : array [1 .. n] of module_type;

initialize
    to s0
    begin
        all i: 1 .. n do
            begin
                init X[i] with body_id;
                X[i].y := 0; { exported variable reference }
            end
    end
```

where the variable "y" is exported by a child module.

Exported variables may be referenced in the **BOOLEAN-EXPRESSION** of a **forone** statement or an **exist-one** expression.

Examples:

```
forone T:transport suchthat T.y = 2
    do —
```

or

```
provided exists Z:network suchthat Z.y = 2
    begin — end
```

7.5 Transition declarations

7.5.1 General introduction

The notion of a transition is central to the functioning of any finite state machine. The informal description of Estelle transitions is given in 5.1; the formal semantics is given in 9.5.1.

7.5.2 Transition

7.5.2.1 Syntax

transition-declaration-part = { transition-declaration } .

```
transition-declaration = "trans" transition-group .

transition-group = +{ clause-group transition-block ";" } .

clause-group = [ provided-clause ]
               ψ[ from-clause ]
               ψ[ to-clause ]
               ψ[ any-clause ]
               ψ[ delay-clause ]
               ψ[ when-clause ]
               ψ[ priority-clause ] .

transition-block = CONSTANT-DEFINITION-PART
                   TYPE-DEFINITION-PART
                   VARIABLE-DECLARATION-PART
                   PROCEDURE-AND-FUNCTION-DECLARATION-PART
                   [ transition-name ] STATEMENT-PART .

transition-name = "name" IDENTIFIER ":" .
```

7.5.2.2 Constraints

A **transition-group** shall be well-formed, as defined below.

Definition: We define the notions of a *well-formed* **transition-group** and of the *scope-region* associated with a transition clause.

(a) The *category* of a transition clause may be one of the following: provided, from, to, any, delay, when, or priority.

(b) If A is a **transition-block**, then A is a well-formed **transition-group**.

(c) If $c_1, \ldots, c_n$ $(n \geq 1)$ are clauses of the same category C and $t_1, \ldots, t_n$ are well-formed **transition-groups**, then $c_1 t_1 \cdots c_n t_n$ is a well-formed **transition-group** if and only if, the following three conditions are true:

(1) a clause of category C shall not appear in $t_1, \ldots, t_n$;

(2) if C is a when-clause category then a delay-clause does not appear in $t_1, \ldots, t_n$;

(3) if C is a delay-clause category then a when-clause does not appear in $t_1, \ldots, t_n$.

The scope-region associated with clause c_i is the **transition-group** $c_i t_i$.

The occurrence of an identifier in the **transition-name** of a **transition-block** of a transition of a **transition-declaration-part** shall constitute its defining-point as a transition-identifier for the region that is the **body-definition** closest-containing the **transition-declaration-part**. When an identifier has a defining-point as a transition-identifier for region A and there is a region B enclosed by A which is a **body-definition**, then region B and all regions enclosed by B shall be excluded from the scope of the defining-point for region A.

7.5.2.3 Informal semantics

Each module instance is an EFSM (Extended Finite State Machine). Each time a module instance is permitted to execute, it executes one enabled simple transition (defined below) from among those defined within the module **transition-declaration-part**. The semantics of these transitions together with the way they are selected for execution is given in 9.5, 9.6.3, and 9.6.5. If the **transition-declaration-part** is empty, then each module and each module instance is called *inactive* (see 5.1).

The syntax allows a transition to be identified by an optional name which is written immediately before the **STATEMENT-PART** of the **transition-block**. Such names may be useful for documentation purposes and in implementation hints for automatic implementation tools. Like comments, they do not change the meaning of a specification.

7.5.2.4 Interpretation rules

In Estelle it is possible to nest or telescope transitions, as in the examples below, where e.g., one **from-clause** applies to two **transition-blocks**. However, formal semantics are defined only for simple transitions, i.e., transitions which are neither nested nor contain an **any-clause**. The first part of this clause defines the mapping from nested transitions to expanded transitions. The second part defines the interpretation rule for the **any-clause**. Applying both parts to a transition yields a set of simple transitions equivalent to the original one.

7.5.2.4.1 Expanding nested transitions

Definitions:

An *expanded transition* is a well-formed **transition-group** that contains exactly one **transition-block**.

An *initialize transition* is an expanded transition that appears after the keyword "initialize".

An *input transition* is an expanded transition that has a **when-clause** in its **clause-group**.

A *spontaneous transition* is an expanded transition that has no **when-clause** in its **clause-group**.

Any **transition-declaration** is a shorthand notation for a sequence of **transition-declarations**, each containing exactly one expanded transition. This equivalence is defined by the following recursive function, **expand**:

Let "trans" t be a **transition-declaration**, where t is a well-formed **transition-group**. Then it follows from the definition of well-formedness that $t = c_1 t_1 \cdots c_n t_n$, where all c_i are of the same category, for $i = 1, \ldots, n$.

To define **expand**, we define three additional functions: **replace**, **insert**, and **expand′**.

The function **replace** replaces "provided otherwise" by a corresponding "provided b", where b is a boolean expression. If there is no "provided otherwise", the function does nothing. Note that from 7.5.5.2, it follows that "provided otherwise", if it occurs, must be c_n.

The definition of **replace** is as follows:

```
if $c_n$ is not “provided otherwise”:
    replace(t) = t
otherwise ($c_n$ is “provided otherwise”):
    there are two cases:
        if n = 1:
            replace($c_1 t_1$) = “provided true” $t_1$
        if n > 1:
            replace($c_1 t_1 \cdots c_n t_n$) =
                $c_1 t_1 \cdots c_{n-1} t_{n-1}$ “provided not” (($b_1$) “or” ($b_2$) “or” $\cdots$ “or” ($b_{n-1}$)) $t_n$,
                    where for $i = 1, \ldots, n-1$, $c_i$ is “provided” $b_i$.
```

The definition of **insert** is as follows:

```
insert(c, “trans” $t_1 \cdots$ “trans” $t_n$ ) = “trans” $ct_1$ “trans” $ct_2 \cdots$“trans” $ct_n$
```

The definition of **expand′** is as follows:

```
if t is expanded:
    expand′(“trans” t) = “trans” t
otherwise:
    recall that $t = c_1 t_1 \cdots c_n t_n$.
    There are two cases:
        if n = 1:
            expand′(“trans” t) = insert($c_1$, expand′(“trans” replace($t_1$)))
        if n > 1:
            expand′(“trans” t) = insert($c_1$, expand′(“trans” replace($t_1$)))
                                  expand′(“trans” $c_2 t_2 \cdots c_n t_n$)
```

The function **expand** can now be defined as:

```
expand(“trans” t) = expand′(“trans” replace(t))
```

Changing the keyword “trans” to “initialize” in the above functions gives the values for the **initialization-part**.

Example:

```
(a) trans
        when ip.m
            from A
                any X : 1..2 do
                    provided E(X)
                        to C
                            begin
                                S11(X); S12(X)
                            end;
                    provided otherwise
                        to C
                            begin
                                S2
                            end;
                from B
                    to C
                        begin
                            S3
                        end;
```

is a shorthand notation semantically equivalent to (b).

```
  (b) trans
          when ip.m
              from A
                  any X : 1..2 do
                      provided E(X)
                          to C
                              begin
                                  S11(X); S12(X)
                              end;
      trans
          when ip.m
              from A
                  any X : 1..2 do
                      provided not E(X)
                          to C
                              begin
                                  S2
                              end;
      trans
          when ip.m
              from B
                  to C
                      begin
                          S3
                      end;
```

7.5.2.4.2 Interpretation of any-clause

If an **any-clause** occurs in an expanded transition then the corresponding **transition-declaration** is only a shorthand notation for a sequence of **transition-declaration**s that contain no **any-clause**. The transitions in this sequence are called *simple*. Each of these **transition-declaration**s contains an expanded transition resulting from the original expanded transition by replacing each applied occurrence of each **IDENTIFIER** in the **IDENTIFIER-LIST** of the **any-clause** by one and the same value from the **ORDINAL-TYPE** of the **any-clause** (see 7.5.9). There shall be one simple, expanded transition for each vector of values of **ORDINAL-TYPE**(s) contained in the **any-clause**. The length of these vectors is given by the number of identifiers in the **IDENTIFIER-LIST** of the **domain-list** of the **any-clause**.

Example:

The first transition in part (b) in the above example is a shorthand for:

```
trans
    when ip.m
        from A
            provided E(1)
                to B
                    begin
                        S11(1); S12(1)
                    end;
trans
    when ip.m
        from A
            provided E(2)
                to B
                    begin
                        S11(2); S12(2)
                    end;
```

7.5.3 To clause

7.5.3.1 Syntax

to-clause = "to" to-element .

to-element = "same" | state-identifier .

7.5.3.2 Constraints

A **to-clause** shall not be used if there is no **state-definition-part** in the **declarations** of the **declaration-part** of the closest-containing **body-definition**.

7.5.3.3 Informal semantics

The **to-element** in the **to-clause** of a **transition-declaration** specifies the next control state following a transition's execution (see 9.6.2(d)). If the **to-element** is **same**, then the control state does not change.

If the **to-clause** is omitted, then the control state does not change.

7.5.4 From clause

7.5.4.1 Syntax

from-clause = "from" from-list .

from-list = from-element { "," from-element } .

from-element = state-identifier | state-set-identifier .

7.5.4.2 Constraints

A **from-clause** shall not be used if there is no **state-definition-part** in the declaration of the closest-containing **body-definition**.

NOTES

1 Elements of a **from-clause** need not be distinct.

2 **State-identifiers** and **state-set-identifiers** may be mixed in a **from-list**.

7.5.4.3 Informal semantics

The **from-list** in a **from-clause** specifies those control states from which a transition may be validly executed (i.e., the **from-clause** is a part of the enabling condition; see 9.6.2(c) and 9.6.3).

If no **from-clause** is present, it is assumed satisfied; i.e., the transition may be executed regardless of the control state.

7.5.5 Provided clause

7.5.5.1 Syntax

provided-clause = "provided" (BOOLEAN-EXPRESSION | "otherwise") .

7.5.5.2 Constraints

The **otherwise** case, if present, shall appear only in a well-formed **transition-group** of the form $c_1 t_1 \cdots c_n t_n$ (see the constraints for a well-formed **transition-group**, 7.5.2.2) as the last clause, c_n.

NOTE — Evaluating the **BOOLEAN-EXPRESSION**, as with all other expressions, has no side effects (see 8.2.5).

7.5.5.3 Informal semantics

The **provided-clause** is a part of the enabling condition of transition. It is satisfied if the **BOOLEAN-EXPRESSION** evaluates to **true**; see 9.6.2(b) and 9.6.3. By the **replace** function, 7.5.2.4, a well-formed **transition-group**:

provided p_1 t_1
provided p_2 t_2
...
provided otherwise t_n

is semantically equivalent to

provided p_1 t_1
provided p_2 t_2
...
provided NOT (p_1 OR p_2 OR $\cdots$ OR p_{n-1}) t_n.

If the **provided-clause** is omitted, it is equivalent to specifying "provided true".

7.5.6 When clause

7.5.6.1 Syntax

```
when-clause = "when" when-ip-reference "." interaction-identifier
                                          [ interaction-argument-list ] .

when-ip-reference = interaction-point-identifier [ "[" ip-index { "," ip-index } "]" ] .

ip-index = CONSTANT | VARIABLE-IDENTIFIER .

interaction-argument-list = "(" interaction-argument-identifier
                            { "," interaction-argument-identifier } ")" .
```

7.5.6.2 Constraints

The occurrence of an **interaction-identifier** in a **when-clause** shall constitute a defining-point of each **interaction-argument-identifier** in the **interaction-argument-list** associated with the **interaction-identifier** for the scope-region associated with the **when-clause** (see 7.5.2.2 for the definition of scope-region).

If the interaction point referenced is associated with some **role-identifier** which in turn is associated with some **channel-identifier**, then the **interaction-identifier** referenced shall be associated with the opposite **role-identifier** that is associated with the same **channel-identifier**.

The **interaction-argument-list** in a **when-clause** is optional, even if the associated **interaction-definition** of the **interaction-identifier** contains arguments. If the **interaction-argument-list** is present, it shall consist of exactly the **interaction-argument-identifiers**, in the same order, whose defining-point for the associated **interaction-definition** is found in the **value-parameter-specification** of the associated **interaction-definition**.

7.5.6.3 Informal semantics

A **when-clause** is a part of the enabling condition of a transition. It is satisfied if the interaction identified by the **interaction-identifier** is at the head of the queue associated with the interaction point indicated by the **when-ip-reference** (see 9.6.2(a) and 9.6.3). The interaction at the head of the queue is dequeued only as part of the execution of the transition.

If the **interaction-argument-list** is not present, then the **when-clause** is semantically equivalent to the same **when-clause** augmented with an **interaction-argument-list** that satisfies the constraint of 7.5.6.2.

7.5.7 Delay clause

7.5.7.1 Syntax

```
delay-clause  =  "delay" "(" ( EXPRESSION "," EXPRESSION
                             | EXPRESSION "," "*"
                             | EXPRESSION )
                   ")" .
```

7.5.7.2 Constraints

The value specified by an **EXPRESSION** of a **delay-clause** shall be of type **INTEGER**.

If a **delay-clause** is specified in a transition of a module, then the **timescale** option of a **specification** shall be specified.

NOTES

1 Evaluating the expressions in a **delay-clause**, as with all other expressions, has no side effects (see 8.2.2 and 8.3.5).

2 A **delay-clause** may be used only for spontaneous transitions; that is, transitions which do not contain a **when-clause**.

7.5.7.3 Interpretation rules

A **delay-clause** "delay (E1)" is semantically equivalent to "delay (E1, E1)"; see 5.3.4.

7.5.7.4 Informal semantics

An informal introduction to the treatment of time in the Estelle model is found in 5.3.5.

To understand the meaning of a **delay-clause**, consider a transition t of a module instance within a system (i.e., the instance is a descendant of a system instance within a specification, see 5.3.4) with the clause "delay(E1, E2)". Describe t as newly enabled if t becomes enabled (see 9.6.3) in a computation step of the system (called a *snapshot* in 5.3.4) but t was not enabled in the previous step, or if t was executed in the previous step.

Once newly enabled, t cannot be executed until it remains enabled for at least E1 time units. More precisely, t may be offered for execution only after E1 consecutive time units of being enabled. Whether it is in fact then executed or not depends on other factors (e.g., priority and non-determinism).

There is a subtle difference in treatment of t in the time between E1 and E2 time units and after E2 time units. Once newly enabled, if t remains enabled but is not fired in each consecutive computation step for E time units, where E1 <= E < E2, then even if t is the one and only enabled transition within a module instance at that moment, t still may or may not be executed. Because of the nondeterminism of the semantics of Estelle, the decision may be regarded as "up to the implementer". Thus, for E2 = *, which is interpreted to mean that the delay time has no upper bound, there is no requirement that the transition ever be executed.

Once newly enabled, if t has been enabled but not fired in each consecutive computation step for E2 time units, then t must be offered by the module instance for execution. Of course, if there are other enabled transitions at this moment, then they are all offered for execution.

The precise description of which transitions are offered for execution in a given state, with respect to the delay values, is given in 9.6.5. The computation scheme is described in 5.3. Together these ensure that a transition with a **delay-clause** obeys the informal description given above.

NOTE — The construct "delay (a, *)" is syntactically unambiguous. Note, however, that if this were to be embedded in a comment, it would terminate the comment, as "*)" is an alternative Pascal comment delimiter. A space between the "*" and the ")" eliminates this possibility.

7.5.8 Priority clause

7.5.8.1 Syntax

```
priority-clause = "priority" priority-constant .

priority-constant = UNSIGNED-INTEGER | CONSTANT-IDENTIFIER .
```

7.5.8.2 Constraints

A **priority-constant** shall be a non-negative integer value.

7.5.8.3 Informal semantics

A **priority-clause** serves to order transitions with respect to **priority-constants** (the lowest non-negative integer is the highest priority). It is one of the elements taken into account while selecting fireable transitions in a state (see 9.6.5) from those enabled (see 9.6.3).

If the **priority-clause** is omitted, the lowest priority is assumed.

7.5.9 Any clause

7.5.9.1 Syntax

```
any-clause  =  "any" domain-list "do"  .


domain-list  =  IDENTIFIER-LIST ":" ORDINAL-TYPE
                { ";" IDENTIFIER-LIST ":" ORDINAL-TYPE }  .
```

7.5.9.2 Constraints

The occurrence of an **IDENTIFIER** in an **IDENTIFIER-LIST** of the **domain-list** of an **any-clause** of a **transition-group** shall constitute its defining-point as a **variable-identifier** for the scope-region associated with the **any-clause** (see 7.5.2.2 for the definition of scope-region).

An **IDENTIFIER** occurring in an **IDENTIFIER-LIST** of a **domain-list** of an **any-clause** shall not be threatened within the scope-region of the **any-clause**.

An **ORDINAL-TYPE** of a **domain-list** of an **any-clause** shall be finite, and its bounds shall be statically known.

7.5.9.3 Informal semantics

A transition in which an **any-clause** occurs is a shorthand for a set of transitions. This set is described in the interpretation rules of 7.5.2.4.2.

7.5.10 Initialization part

The **initialization-part** of a **module-body-definition** defines all initial states of the EFSM represented by the module (see 9.6.3).

7.5.10.1 Syntax

```
initialization-part  =  { "initialize" transition-group }  .
```

7.5.10.2 Constraints

Only a **to-clause** or a **provided-clause** shall appear in the **transition-group** of the **initialization-part**.

Same shall not be used in the **to-list** of an **initialization-part**.

See the constraints for a **transition-group** given in 7.5.2.2.

If there is a **state-definition-part** in the **declarations** of a **declaration-part** of a **body-definition**, then the **initialization-part** of the **body-definition** shall be non-empty, and each **transition-block** of the **transition-group** of the **initialization-part** shall be in the scope-region of a **to-clause** (see 7.5.2.2 for the definition of scope-region).

7.5.10.3 Informal semantics

The **initialization-part** defines a procedure to be executed only once when a module instance is created (7.6.1).

Only one **transition-block** is executed when the module is initialized, even if more than one is enabled (nondeterminism). The semantics for nondeterminism is the same as for transitions.

The **to-element** in the **to-clause** of an **initialization-part** specifies the initial control state (see 9.6.2(d)).

7.6 Estelle statements

Module instances may be created, released and terminated, and their interaction points may be bound and unbound in the operations described below. These operations may occur in initialization and transition parts.

7.6.1 Module instance creation

The creation of and initialization of a module is performed with an **init-statement**.

7.6.1.1 Syntax

```
init-statement =  "init" module-variable "with" body-identifier
                  [ "(" actual-module-parameter-list ")" ] .


actual-module-parameter-list =  actual-module-parameter { "," actual-module-parameter } .


actual-module-parameter =  EXPRESSION .
```

7.6.1.2 Constraints

The **header-identifier** which is the type of the **module-variable** referenced in an **init-statement** shall be identical to the **header-identifier** in the **module-body-definition** referenced by the **body-identifier** in the **init-statement**.

The **actual-module-parameter**(s) contained in an **init-statement** shall be compatible and in corresponding order with the **parameter-list** contained in the **module-header-definition** for the **module-variable** referenced.

7.6.1.3 Informal semantics

The execution of an **init-statement** selects a particular body (**body-identifier**) for the module instance and initializes the module instance. An **init** operation assigns a value to the **module-variable** referenced and assigns the values of the elements of the **actual-module-parameter-list** to the elements the formal parameter list defined by the **parameter-list** contained in the **module-header-definition** for the **module-variable** referenced. The module instance will be in one of the initial states that results from executing a transition from its **initialization-part** (7.5.10) if the **initialization-part** is non-empty. If the **initialization-part** is empty, then the instance exists in a preinitial state, as defined in 9.4.5.2.

After execution of the **init-statement**, the **module-variable** identifies the module instance.

The execution of an **init-statement** includes the execution of a transition from the **initialization-part** (see 7.5.10) of a child module whose instance is being created, i.e., the new instance is created in one of its initial states (for details, see 9.6.6.1).

If an **init-statement** references a **module-variable** which already identifies a module instance, a new module instance is created and the **module-variable** referenced is assigned a new value. Access to the former module instance is lost unless the value of the **module-variable** referenced in an **init-statement** is saved in another module variable (of the same type) by execution of an **assignment-statement**. However, access to the module instance may be recovered by the use of an **all-statement** or a **forone-statement** (see 7.6.9 and 7.6.10.)

7.6.2 Release and termination of module instances

7.6.2.1 Syntax

```
release-statement  =  "release" module-variable .


terminate-statement  =  "terminate" module-variable .
```

7.6.2.2 Constraints

NOTE — A module may release or terminate a child module; it may not release or terminate itself or a sibling module.

7.6.2.3 Informal semantics

The result of a **release-statement** is that first, all external interaction points of the module instance identified by the **module-variable** which have been attached or connected are detached or disconnected (see 7.6.5 and 7.6.6); and second, the module instance and all its descendent instances are released and are no longer available. The value of all **module-variable**s which identified the released instance are undefined, as though they were never initialized.

The result of a **terminate-statement** is that first, all external interaction points of the module instance identified by the **module-variable** which have been attached or connected are detached (an auxiliary statement, simple-detach, defined in 9.6.6.2.4, is used) and disconnected (see 7.6.5); and second, the module instance and all its descendent instances are terminated and are no longer available. The value of all **module-variable**s which identified the terminated instance are undefined, as though they were never initialized.

The only difference between a **release-statement** and a **terminate-statement** is that the external interaction points of the module instance identified by the **module-variable** that have been attached are detached in a different manner. In the case of a **release-statement**, the contents of (some of) the queues of the module initiating the statement may be changed as a result of the semantics of the detach operation. In the case of the **terminate-statement**, no queue of the module initiating the statement is changed, reflecting the difference between the detach operation and the simple-detach operation.

The sequence of statements

```
detach X;
terminate X
```

is semantically equivalent to

```
release X
```

7.6.3 Connect operation

7.6.3.1 Syntax

```
connect-statement = "connect" connect-ip "to" connect-ip .
```

7.6.3.2 Constraints

The two interaction points referenced in a **connect-statement** shall be declared using identical **channel-identifiers** and opposite **role-identifiers** within that channel.

A **connect-statement** shall not attempt to connect an interaction point that is currently bound by the module executing the **connect-statement**.

NOTE — Table 2 (see 7.6.7) summarizes valid combinations of references to pairs of interaction points within a **connect-statement**.

7.6.3.3 Informal semantics

After the execution of a **connect-statement**, the interaction points referenced are bound. Any interactions output through an interaction point are received at the interaction point to which the former is connected (or its descendant if that interaction point is attached). It is possible that the external interaction point of a child which is connected by its parent was previously (or will be) attached by the child itself to an external interaction point of one of its own children. Interactions received are queued according to the queuing discipline specified for the interaction point of the receiving module.

An external interaction point of a module instance which is connected cannot be attached, at the same time (see 9.4.5.1 and 9.6.6.3), to an external interaction point of its parent instance (but may be attached to an interaction point of its children instances). If a **connect-statement** is executed by a module instance it may connect:

— two external interaction points of children instances;

— two internal interaction points of itself;

— one of its internal interaction points with an external interaction point of a child instance.

The details are in 9.6.6.3 and within the well-formedness conditions of a state in 9.4.5.1.

7.6.4 Attach operation

7.6.4.1 Syntax

```
attach-statement = "attach" external-ip "to" child-external-ip .
```

7.6.4.2 Constraints

The **channel-identifier**s and the **role-identifier**s of the interaction points referenced by an **attach-statement** shall be identical.

An **attach-statement** shall not attempt to attach an interaction point that is currently bound by the module executing the **attach-statement**.

NOTE — Table 2 (see 7.6.7) summarizes valid combinations of references to pairs of interaction points within an **attach-statement**.

7.6.4.3 Informal semantics

After the execution of an **attach-statement**, the first interaction point referenced is bound to the second interaction point. In the constraint above it is important to note that the first interaction point may be currently bound by an action of the parent of the module executing the attach operation, and the second may be bound by an action of a child. An interaction point at the end of a sequence of bound interaction points, where one of the bindings must be from a connect operation, is defined to be a *connection endpoint*. In 9.5.3, such a sequence is referred to as *linked*.

The effect of execution of **output** statements by a module at a connection endpoint is that the interactions are appended to the queue of the interaction point bound to the opposite connection endpoint.

When an **attach-statement** is executed, interactions present in the queue of the external interaction point of the module instance issuing the statement are removed from this queue. In case of a common queue, only those interactions that came through this external interaction point are removed. These are appended to the queue of the external interaction point of the lowest level descendent module instance which is attached, by a sequence of attached interaction points, to the second interaction point in the **attach-statement**. Details are in 9.6.6.2 and within the well-formed conditions of a state in 9.4.5.1.

Note that the interactions for two interaction points whose queuing option is **individual queue** within a parent module may be combined into a common queue of a child through **attach** operations.

7.6.5 Disconnect operation

7.6.5.1 Syntax

```
disconnect-statement  =  "disconnect" ( connect-ip  | module-variable ) .
```

7.6.5.2 Constraints

The interaction point referenced by the **connect-ip** shall be currently bound by a connect.

NOTE — A summary of valid uses of the **disconnect-statement** is found in Table 2 (see 7.6.7).

7.6.5.3 Informal semantics

The result of a **disconnect-statement** in the first case is that the interaction point given by the **connect-ip** and the one with which it has been connected are unbound.

The result of a **disconnect-statement** in the second case is that all the connected external interaction points of the child module instance identified by the **module-variable** are disconnected. (See 9.6.6.3 for formal definitions of these operations.)

NOTE — When an interaction point is disconnected, the interactions queued for it remain in the queue. Therefore it is possible for a module to process interactions that were queued before the interaction point was disconnected.

7.6.6 Detach operation

7.6.6.1 Syntax

```
detach-statement  =  "detach" ( external-ip  | child-external-ip  | module-variable ) .
```

7.6.6.2 Constraints

The module executing the **detach-statement** shall be the same module that attached the interaction point, i.e., the parent.

NOTE — A summary of valid uses of the **detach-statement** is found in Table 2 (see 7.6.7).

7.6.6.3 Informal semantics

The first syntactic alternative references an external interaction point of the module executing the **detach** operation, whereas the second alternative references an external interaction point of a child. The effects of the operation are identical regardless of which syntactic alternative is used.

The result of a **detach-statement** in the first two syntactic alternatives is that the referenced interaction point and the one (and only one) to which it has been attached are unbound.

Assume ip1 is the relevant external interaction point of the module executing the **detach-statement**; and assume ip2 is an external interaction point of the lowest level descendant module which is attached, by a sequence of attached interaction points, to ip1. When the **detach-statement** is executed, certain of the interactions present in the queue assigned to ip2 are removed from this queue and appended to the queue assigned to ip1: these are precisely the interactions that were appended to the queue passing through both ip1 and ip2. Thus, interactions initiated by a module at a lower level in the hierarchy are not moved to the module executing the **detach** operation.

The result of the **detach-statement** in the third syntactic alternative is that all the attached, external interaction points of the child module instance identified by the **module-variable** are detached. (See 9.6.6.2 for the formal definition.)

7.6.7 Summary of binding operations

Examples:

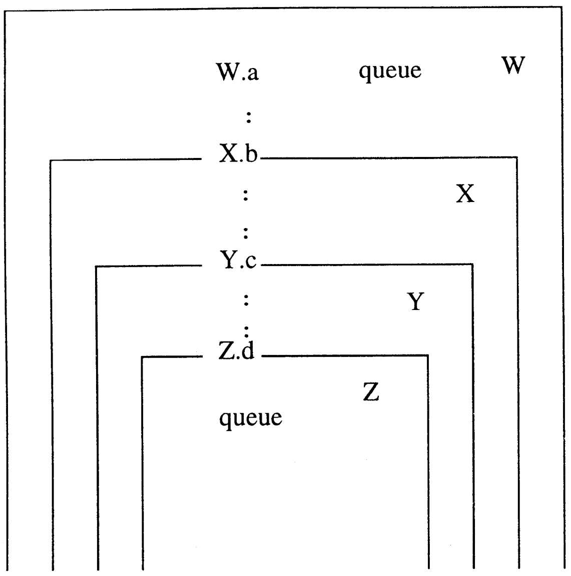
Figures 2 and 3 summarize, by example, the actions of the **connect**, **attach** and **detach** operations given in the following sequence of statements contained in modules "W", "X", "Y", and "Z" with interaction point "a" internal to module "W" and "b", "c", and "d" external interaction points of "X", "Y", and "Z" respectively.

Statement	**Module of Execution**
connect a to X.b;	(* within W *)
attach b to Y.c;	(* within X *)
attach c to Z.d;	(* within Y *)

In Figures 2 and 3, the modules with access to queued interactions are indicated by the word "queue".

If subsequently a **detach** operation is executed by module "X" in either of the following forms,

```
detach b;
detach Y.c;
```



Legend:

W, X, Y, Z -
module identifiers
a, b, c, d -
interaction point identifiers

. interaction point
: bound interaction points

Binding of interaction points and access to queued interactions after connect and attach operations.

Figure 2

interaction points “b” and “c” are unbound and the interactions of the queue at “d” that passed through “b” (not those that entered through “c”) are immediately moved to the end of the queue of module “X” for interaction point “b” and their order is preserved. Subsequent interactions initiated by module “W” are received by module “X” and are appended to its queue. This is the situation depicted in Figure 3.

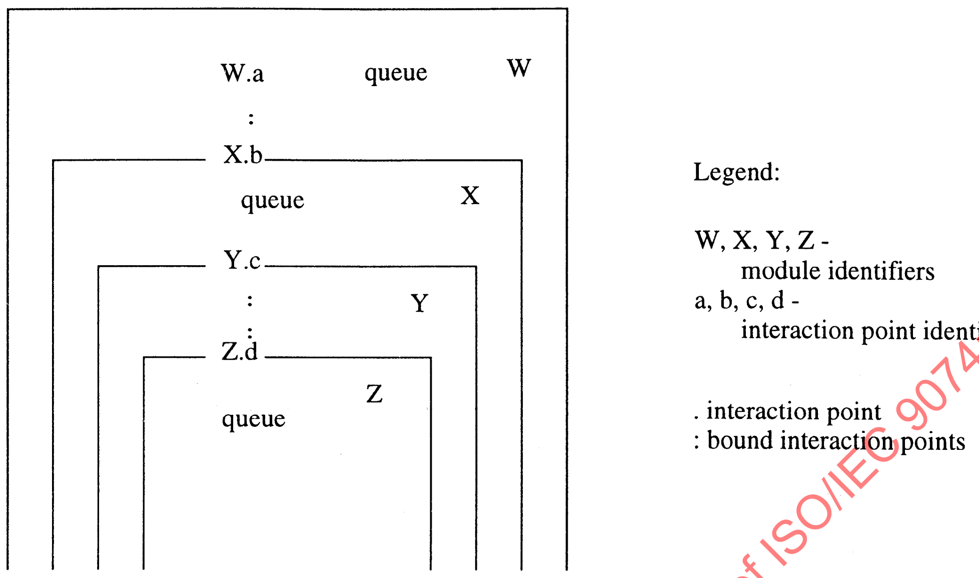
Table 2 lists all valid combinations interaction point references (operands) which may appear in the **attach**, **connect**, **detach**, and **disconnect** statements. The columns labelled first and second operand identify whether the interaction point is that of a parent or child module. The columns labelled interaction point type indicate whether the interaction point is internal or external to the module executing the operation.

7.6.8 Output statement

7.6.8.1 Syntax

output-statement = “output” interaction-reference [ACTUAL-PARAMETER-LIST] .

interaction-reference = interaction-point-reference “.” interaction-identifier .


W.a
queue
W
:
X.b
queue
X
Y.c
:
Y
:
Z.d
Z
queue
Legend:
W, X, Y, Z - module identifiers
a, b, c, d - interaction point identifiers
. interaction point
: bound interaction points

Binding of interaction points and access to queued interactions after a detach operation.

Figure 3

7.6.8.2 Constraints

The parameters of the **ACTUAL-PARAMETER-LIST** shall be compatible with those of the **interaction-definition**.

A module which has bound one of its interaction points with an **attach-statement** shall not subsequently reference the interaction point in an **output-statement** while the interaction point remains attached.

7.6.8.3 Informal semantics

The result of an **output-statement** is that the interaction, defined by the **interaction-identifier** along with its actual parameters, if any, will be appended to the queue assigned to the [other] connection endpoint, which is an interaction point linked (see 9.5.3) to the interaction point referenced by the **interaction-reference** in the **output-statement**.

It is important to note that outputs made through an external interaction point may be observed in the target queue only after the whole issuing transition is completed. This is an important assumption concerning the atomicity of transitions.

See 9.6.6.5, 9.5.3, and 9.5.4 for the formal definition of the **output-statement**.

NOTE — An output statement which is contained in the **initialization-part** of a module should not reference an external interaction point of the module. If it did, it would result in an interaction's being discarded because the interaction point cannot be linked to another interaction point during module initialization. This is because only a parent module may initialize a child module, and only subsequently may the parent bind the child's external interaction points.

Table 2 — Valid Uses of Binding Operators

Operation Name	First Operand	Interaction Point Type	Second Operand	Interaction Point Type
Attach	parent	external	child	external
Connect	parent	internal	child	external
Connect	child	external	parent	internal
Connect	child	external	child	external
Connect	module(1)	internal	module(1)	internal
Detach	parent	external	None	
Detach	child	external	None	
Detach	child module(2)	all(3)	None	
Disconnect	child	external	None	
Disconnect	parent	internal	None	
Disconnect	child module(2)	all(4)	None	

Notes for Table 2:

(1) Any module may connect two of its own internal interaction points that have complementary roles.

(2) The module referenced by a **module-variable**.

(3) All external interaction points bound by an attach operation at the time when the detach operation is executed are unbound for the module referenced (child). Since the detach operation may not unbind interaction points bound with a connect operation, separate disconnect operations may be required.

(4) All external interaction points bound by a connect operation at the time when the disconnect operation is executed are unbound for the module referenced (child). Since the disconnect operation may not unbind interaction points bound with an attach operation, separate detach operations may be required.

7.6.9 All statement

The **all-statement** is a repetitive statement which allows iteration over an ordinal-type or over a set of module instances.

7.6.9.1 Syntax

```
all-statement =  "all" ( domain-list | module-domain )
                 "do" STATEMENT .


module-domain =  IDENTIFIER ":" header-identifier .
```

7.6.9.2 Constraints

The occurrence of an **IDENTIFIER** in either the **domain-list** or the **module-domain** of the an **all-statement** shall constitute its defining-point as a **variable-identifier** for the region that is the **all-statement**.

An **ORDINAL-TYPE** of a **domain-list** of an **all-statement** shall be finite, and its bounds shall be statically known.

7.6.9.3 Informal semantics

The bounds of a domain are evaluated once and are not affected by execution of the **STATEMENT** within the scope of the **all-statement**.

NOTE — These conditions are similar to the conditions applied to the control variable of a **for** loop in Pascal.

The result of an **all-statement** is the execution of the **STATEMENT** for either:

(a) all vectors of values of ordinal type(s) given in the **domain-list** (the length of these vectors is given by the number of **variable-identifiers** declared in the **domain-list**); or

(b) all children instances whose header definitions are identified by the **header-identifier** in the **module-domain**.

If the domain is empty, the statement following **do** is not executed.

The order of execution is arbitrary.

NOTE — Results may depend on the order of execution if the statement includes a procedure that has side effects. Therefore it is strongly recommended this statement not be used if its result depends on the order.

See 9.6.6.6.3 for the formal definition.

Example: The **all-statement** and module instances:

The **all-statement** can be used for referencing module instances in two different ways.

The domain identified by a **header-identifier** of a **module-domain** may contain explicit access to an array of module instances (a static domain). After evaluating the domain, the **do** statement is executed for each element of the array. For example,

```
modvar x: array [1 .. n] of module_type;
initialize
     to s0
          begin
               for i := 1 to n do
                    init x[i] with body_id;
          end;
trans
     from s0 to s1
          begin
               all i: 1 .. n do
                    connect x[i].external_ip to — ;
          end;
     from s1 to s0
          begin
               all i: 1 .. n do
                    release x[i];
          end;
```

The domain identified by the **header-identifier** of a **module-domain** may consist solely of implicit access to all module instances in the dynamic domain of modules of some module type. The domain defines a single identifier of a given module type. The **do** statement is executed for each module instance of that type. For example,

```
modvar T: transport;
initialize
     to s0
          begin
               for i := 1 to n do
                    init T with body_id;
          end;
trans
     from s0
          to s1
               begin
                    all t: transport do
                         connect t.external_ip to — ;
               end;
     from s1
          to s0
               begin
                    all t:transport do
                         release t;
               end;
```

7.6.10 Forone statement

7.6.10.1 Syntax

```
forone-statement = "forone" ( domain-list | module-domain )
                   "suchthat" BOOLEAN-EXPRESSION
                   "do" STATEMENT
                   [ "otherwise" STATEMENT ] .
```

7.6.10.2 Constraints

The occurrence of an **IDENTIFIER** in either the **domain-list** or the **module-domain** of a **forone-statement** shall constitute its defining-point as a **variable-identifier** for the region that is the **forone-statement**.

An **ORDINAL-TYPE** of a **domain-list** of a **forone-statement** shall be finite, and its bounds shall be statically known.

An **IDENTIFIER** in the **domain-list** of a **forone-statement** shall not have an applied occurrence in the **otherwise** clause of the **forone-statement**.

7.6.10.3 Interpretation rules

The **forone-statement** without the **otherwise** clause is semantically equivalent to the one with the **otherwise** clause containing only an **EMPTY-STATEMENT**.

7.6.10.4 Informal semantics

The bounds of a domain are evaluated once and are not affected by any evaluation of the **BOOLEAN-EXPRESSION** within the scope of the **forone-statement**.

NOTE — These conditions are similar to the conditions applied to the control variable of a **for** loop in Pascal.

If the **BOOLEAN-EXPRESSION** introduced by **suchthat** evaluates to **true** for at least one element of the set of either:

(a) vectors of values of **ORDINAL-TYPE**(s) given in the **domain-list**, or

(b) all children instances whose header definitions are identified by the **header-identifier** of the **module-domain**,

then the **STATEMENT** introduced by the **do** keyword is executed for one element of the set of values defined by (a) or (b) for which the **BOOLEAN-EXPRESSION** evaluates to **true**.

If the **BOOLEAN-EXPRESSION** introduced by **suchthat** evaluates to **false** for all such elements, or if there are no such elements, then the **STATEMENT** contained in the **otherwise** clause, if present, is executed.

The **forone-statement** without the **otherwise** part is semantically equivalent to the one with the **otherwise** part containing an **EMPTY-STATEMENT**.

NOTE — Implicit in the semantics of the **forone-statement** is a search through a set of values. The order of this search is not specified, but because evaluating the **BOOLEAN-EXPRESSION**, as with all other expressions, has no side effects (see 8.2.5.1) the semantics of the **forone-statement** does not depend on the order in which the search is carried out.

When **forone-statements** are nested, each **otherwise** clause is associated with a **forone-statement** in exactly the same way as the **else-part** is associated with an **if-statement** in Pascal (see 6.8.3.4 of Annex C).

7.6.11 Exist expression

7.6.11.1 Syntax

```
exist-one =  "exist" ( domain-list | module-domain)
             "suchthat" FACTOR .
```

7.6.11.2 Constraints

The occurrence of an **IDENTIFIER** in either the **domain-list** or the **module-domain** of the an **exist-one** expression shall constitute its defining-point as a **variable-identifier** for the region that is the **exist-one** expression.

An **ORDINAL-TYPE** of a **domain-list** of an **exist-one** expression shall be finite, and its bounds shall be statically known.

7.6.11.3 Informal semantics

An **exist-one** expression is a boolean expression that is satisfied if there is at least one element in the set of either:

(a) vectors of values of **ORDINAL-TYPE**(s) given in the **domain-list**, or

(b) all children instances whose header definitions are identified by the **header-identifier** of the **module-domain**,

satisfying the **BOOLEAN-EXPRESSION** (see 9.6.6.6.1).

7.7 Reserved words

The following reserved words are introduced in the previous grammar rules.

7.7.1 Syntax

```
key-words =     "activity"        | "all"            | "any"
              | "attach"          | "body"           | "by"
              | "channel"         | "common"         | "connect"
              | "default"         | "delay"          | "detach"
              | "disconnect"      | "exist"          | "export"
              | "external"        | "forone"         | "from"
              | "individual"      | "init"           | "initialize"
              | "ip"              | "module"         | "modvar"
              | "name"            | "otherwise"      | "output"
              | "primitive"       | "priority"       | "process"
              | "provided"        | "pure"           | "queue"
              | "release"         | "same"           | "specification"
              | "state"           | "stateset"       | "suchthat"
              | "systemactivity"  | "systemprocess"  | "terminate"
              | "timescale"       | "trans"          | "when" .
```

7.7.2 Constraints

No **IDENTIFIER** shall have the same spelling as any **key-word**.

No **DIRECTIVE** shall have the same spelling as any **key-word**.

8 Extensions and restrictions to ISO Pascal

This clause summarizes extensions and restrictions to ISO Pascal. Annex C is based on ISO/IEC 7185:1990 and defines the subset of Pascal used by this International Standard, subject to the extensions and restrictions given in this clause. Changes to the grammar of Pascal appear in this clause and take precedence over nonterminal definitions given in Annex C.

8.1 Simple changes to Pascal syntax

The following modifications to the Pascal grammar given in Annex C are necessary to include Estelle constructs defined in clause 7.

In the following rules, the construct "—" refers to the right hand side of the corresponding definition of the lower case nonterminal in ISO Pascal (see Annex C).

8.1.1 Syntax

```
COMPONENT-VARIABLE = — | exported-variable .
```

```
FACTOR = — | exist-one .

LETTER = — | "_" .

REPETITIVE-STATEMENT = — | all-statement .

RESULT-TYPE = TYPE-IDENTIFIER .

SIMPLE-STATEMENT = — | attach-statement
                     | connect-statement
                     | detach-statement
                     | disconnect-statement
                     | init-statement
                     | output-statement
                     | release-statement
                     | terminate-statement .

STRING-CHARACTER = any-character-specified-in-ISO/IEC-646 .
```

NOTE — A **STRING-CHARACTER** is restricted to those characters defined in ISO/IEC 646.

```
STRUCTURED-STATEMENT = — | forone-statement .

WORD-SYMBOL = — | key-words .
```

8.2 Extensions

8.2.1 Integers and real numbers

Integer and real numbers are considered to be the integers and the real numbers in a mathematical sense. Implementation dependent constraints of maximum size and precision of real numbers are not specified in this International Standard.

8.2.2 Functions and procedures

Functions may return structured data types; e.g., arrays and records. All function-declarations shall be demonstrably pure, as defined in 8.2.5.1.

Functions and procedures shall not reference non-Pascal objects (e.g., module variables, interaction points, interactions, or states) in accordance with the scope rules expressed in clause 7 (see e.g., 7.3.9.2, 7.3.6.2, 7.3.8.1, 7.3.10.2). As a consequence, the Estelle statements **all**, **forone**, and **exist** within a **PROCEDURE-BLOCK** or a **FUNCTION-BLOCK** shall be used with a **domain-list** limited to **ORDINAL-TYPE**. As a further consequence, the Estelle statements **init**, **release**, **connect**, **disconnect**, **attach**, **detach**, and **output** shall not be used in a **PROCEDURE-BLOCK** or a **FUNCTION-BLOCK**.

8.2.3 Implementation defined elements

8.2.3.1 General

In constructing a formal specification, it may be convenient to permit partial specification in various ways. Estelle provides facilities for this purpose within constant declarations, type declarations and function declarations as specified in 8.2.3.2, 8.2.3.3, and 8.2.4.

8.2.3.2 Constants

8.2.3.2.1 Syntax

```
CONSTANT-DEFINITION = — | IDENTIFIER "=" "any" TYPE-IDENTIFIER .
```

8.2.3.2.2 Informal semantics

The **any** construct permits the type of a constant to be defined without attributing a specific value to it. Use of the **any** construct permits a specification to be statically checked for type compatibility before the values of such constants are known or can be supplied. This construct is included in the language syntax for the convenience of developers and users of support tools.

An Estelle specification containing an occurrence of the **any** construct is statically checkable for type compatibility but is not well-formed. Such a specification cannot have meaning attributed to it by application of the semantic rules in clause 9. Note, however, that once all such constant definitions have been replaced, the formal semantics is defined for the resultant specification (see clause 9).

8.2.3.3 Types

8.2.3.3.1 Syntax

TYPE-DEFINITION = — | IDENTIFIER "=" "..." .

8.2.3.3.2 Informal semantics

The "..." construct permits a type-identifier to be declared without attributing a specific type to it. Use of this construct permits limited type checking of specification texts before the precise definitions of such type-identifiers are known or can be supplied. No assumptions can be made about the types of variables whose declarations include type-identifiers associated with the "..." construct and such variables shall not appear in syntactic contexts where type compatibility rules do not permit the occurrence of a variable of arbitrary type.

An Estelle specification containing an occurrence of the "..." construct is statically checkable subject to the limitations imposed by making no assumptions about the types of relevant variables. However, such a specification is not well-formed and cannot have meaning attributed to it by application of the semantic rules in clause 9. Note, however, that once all type definitions of the form "..." have been replaced, the formal semantics is defined for the resultant specification (see clause 9). Note futher that simple assumptions may be made about variables declared to be of type "...". For instance, two variables declared to be the same type may be compared, even if that type is "...".

8.2.4 Directives

The directives **forward**, **external**, and **primitive** shall be required directives. No **directive** shall have the same spelling as any **WORD-SYMBOL**.

8.2.4.1 Syntax

directive = letter { letter | digit } .

NOTE — Directives are intended for use by processors of Estelle and permit partial specifications to be written that can be separately evolved and independently combined.

8.2.4.2 Constraints

When an **identifier** has a defining-point as a **procedure-identifier** or **function-identifier** for region A, and the **procedure-heading** or **function-heading** closest-containing the identifier does not contain the directive **primitive**, and there is a region B enclosed by A which is a **body-definition**, then region B and all regions enclosed by B shall be excluded from the scope of the defining-point for region A.

8.2.4.3 Informal semantics

The directive **primitive** is used to indicate that the block of a function or procedure is not given in the specification text and may in general be available only at the time of activation. This is in contrast to the directive **external**, which indicates that the block is given in another source and may be included from that source prior to activation.

An Estelle specification containing the directive **primitive** is well-formed, but meaning may be attributed to it only if a rigorous, implementation independent (e.g., mathematical) definition of the relevant block is supplied by the specifier.

The block of a procedure or function may be specified in a separate description, in which case it is designated by the required directive **external**.

The scope of **primitive** functions and procedures is global from their defining-point.

8.2.5 Pure procedures and functions

8.2.5.1 Demonstrably pure

A type shall be designated as *pointer-containing* (cf. 7.3.4.2) if it is a **pointer-type** or it is a **structured-type** possessing a **component-type** that is pointer-containing.

A declaration of a procedure or function shall be designated as *demonstrably pure* if and only if the following conditions are satisfied:

(a) if it is a declaration of a function, then it does not contain a **variable-parameter-specification**;

(b) it does not contain a **value-parameter-specification** that contains a **type-identifier** denoting a pointer-containing type;

(c) its **procedure-block** or **function-block** does not contain any statement threatening a variable whose scope includes a region not included within the region of that block;

(d) its **procedure-block** or **function-block** does not contain any applied occurrence of either a **function-identifier** or a **procedure-identifier** whose declaration is not demonstrably pure;

(e) no **function-designator** or **procedure-statement** closest-containing an applied occurrence of its **function-identifier** or **procedure-identifier** for which the corresponding declaration is not demonstrably pure.

(f) its **procedure-block** or **function-block** does not contain a variable whose scope includes a region not included within the region of that block and whose type is pointer-containing.

Every **function-declaration** shall be demonstrably pure.

Remark: The intention of the preceding clause is to prohibit side effects arising from the evaluation of functions. It states strong, sufficient conditions to ensure this.

8.2.5.2 Syntax

PROCEDURE-HEADING = — | "pure" "procedure" IDENTIFIER [FORMAL-PARAMETER-LIST] .

PROCEDURE-IDENTIFICATION = — | "pure" "procedure" PROCEDURE-IDENTIFIER .

8.2.5.3 Constraints

Procedures whose **procedure-declaration** contains the reserved word **pure** shall be demonstrably pure, except for assignment to **var** parameters. In particular, only a local variable or a variable declared as **var** parameter in the heading, may be assigned values, either directly, or indirectly via a function or procedure.

If a **procedure-heading** contains the keyword **pure**, then the corresponding **procedure-identification** shall also contain the keyword **pure**.

8.2.5.4 Informal semantics

Table 3 summarizes the constraints regarding the combined use of functions and procedures with the pure and nonpure attributes.

Table 3 — Use of Pure and Non-Pure Attributes

Cross references between Procedures or Functions with pure/nonpure attributes		
Defining-Point	Makes Reference to	Allowed
nonpure	nonpure	yes
nonpure	pure	yes
pure	pure	yes
pure	nonpure	no

NOTE — only the last two entries apply to functions.

8.2.6 Expression

Module variables may be compared to other module variables.

8.2.6.1 Syntax

EXPRESSION = — | module-variable RELATIONAL-OPERATOR module-variable .

8.2.6.2 Constraints

The relational operators used in an **EXPRESSION** referencing a **module-variable** shall be either equality ("=") or inequality ("<>").

Both **module-variable**s shall be declared with the same **header-identifier**.

8.2.6.3 Informal semantics

If the two module variables are defined and refer to the same module instance, then they are equal; otherwise they are not equal.

8.2.7 Assignment operation

Module variables may be referenced in an assignment statement.

8.2.7.1 Syntax

```
ASSIGNMENT-STATEMENT = — | module-variable ":=" module-variable .
```

8.2.7.2 Constraints

The **module-variable**s of an **ASSIGNMENT-STATEMENT** shall both be declared with the same **header-identifier**.

8.2.7.3 Informal semantics

The value attributed to the **module-variable** on the left-hand-side of the assignment statement is set to the value attributed to the **module-variable** on the right-hand-side of the assignment statement.

8.3 Restrictions

There are several restrictions to ISO Pascal [ISO/IEC 7185:1990] for the purposes of this International Standard. The required changes to effect these restrictions have been made to Annex C, which is a part of this International Standard. To aid the reader, these changes are outlined here, but the remainder of clause 8.3 shall be regarded as a NOTE.

Briefly the changes are these:

— All level 1 conformance requirements are excluded. In particular, use of conformant arrays in Estelle specifications is excluded.

— Use of label and goto statements are restricted in Estelle specifications.

— No use of program statements, read or write statements, or file types is permitted.

8.3.1 Errors

The specification of error handling by a "processor" is excluded in this International Standard; specifically, all related statements in the Pascal Standard and their summary in Appendix D are excluded. The term "error" is used as defined in 3.3.

8.3.2 File manipulation

The required procedures of Pascal **get**, **put**, **read**, **readln**, **write**, **writeln**, **eof**, and **eoln** may not be used in Estelle specifications.

References to **read-parameter-list**, **readln-parameter-list**, **write-parameter-list**, and **writeln-parameter-list** have been removed from Annex C.

References to **file-type** and the predefined type **text** have been removed from Annex C.

8.3.3 Label declarations and goto statements

Label declarations are restricted to functions and procedures. Labels may be associated only with (an empty statement immediately before) the **end** of a procedure or function (see Annex C, 6.8.1). Goto statements, although permitted in Estelle specifications, are limited by the following restrictions:

— goto statements shall be used only in procedures and functions, and

— goto statements shall only branch forward to the final **end** statement of a procedure or function, thereby causing the flow of execution to return to the place where the procedure or function was referenced.

NOTE — With these restrictions, all goto's effectively act as "return" statements found in other languages. See Annex C, 6.8.2.4, points (c) and (d).

8.3.4 Program statement

The Pascal nonterminals **program**, **program-block**, **program-heading**, and **program-parameters** have been excluded from the grammar in Annex C.

8.3.5 Expressions and functions

NOTE — As noted in 8.2.2, all Estelle functions must be demonstrably pure. Demonstrably pure functions have no side effects, thus the value of an expression does not depend on the order in which its components are evaluated. Similarly, the values of expressions in a list of index-expressions or actual parameters do not depend on the order in which the elements of the list are evaluated.

8.3.5.1 Constraints

All operators in expressions shall denote the corresponding mathematical operators naturally extended so that they yield undefined results if any of their operands does not denote a defined value at the time of its use.

9 Semantics of Estelle constructs

The general model of the behavior of an Estelle specification given in clause 5 assumes that the semantic description of a single module instance is known. This clause details the semantics of the constructs that characterize a single module instance. Thus this clause complements clause 5. In particular the following concepts are defined: enabled transition, fireable transition, global instantaneous description, and the extension of a transition local interpretation on the set of global instantaneous descriptions. These are necessary to understand properly the formal parts of clause 5 (e.g., 5.3).

The following general assumptions are made:

— All specifications are assumed correct with respect to the context free grammar and the context sensitive constraints given in clause 7. This means that erroneous specifications are not considered. Note that such erroneous specifications may be discovered statically.

— The denotation of an Estelle specification shall be defined by the mathematical constructions of this clause and 5.3. The formal existence of such a denotation shall be dependent upon the existence of denotations of Pascal constructs used in such a specification, provided that such denotations specify corresponding partial functions assumed in this clause to be meanings of Pascal language constructs.

— An Estelle specification may be parameterized in some ways. One way is by introducing the construct of **any** constant of certain type (see 8.2.3.2). Another way is by using the "..." type construct (see 8.2.3.3). In these cases a given text actually defines a collection of specifications, each one identified by particular values substituted for the parameters. Similarly, some procedures or modules may be declared as **external**; i.e., their definition exists elsewhere (see 7.3.7). However they are considered as known or present in a given specification at the moment of describing the semantics; i.e., the **external** construct constitutes a pure inclusion mechanism of the language. To summarize this point, parameters of the form described above and external modules and procedures are treated as a convenient way of writing a collection of specifications. The semantics defined here are only for *complete* specifications, where each parameter has a concrete value assigned to it and each **external** object is available.

— All specifications are assumed interpreted by the interpretation rules of clause 7. This means, for example, that the semantics deals only with simple transitions; thus, for example, each **any** clause and each **provided otherwise** clause has already been interpreted.

Notational conventions used are either self-explanatory mixtures of mathematical and programming notations or are explained when introduced. For a summary of conventions used, see 6.2.

9.1 General scheme of definitions

To define the semantics of a module, each instance of the module must be characterized. The following is the general scheme of this definition:

(a) All definitions are based on some primitives (9.2). A module's semantics is defined with respect to the notion of an external context environment (9.3). The latter consists, roughly speaking, of global information which is available when an instance of a module is being defined, i.e., an inherited type, constant and channel definitions, and primitive functions and procedures.

(b) For each module-header M and for each module-body-definition B for the header M declared within an external context E, given by the embodying bodies of a specification, all necessary elements characterizing the module instance (of M and B in the environment E) as a finite control state transition system are given.

All different instances having the same characteristics (i.e., being interpretations of the same module header M, the same body B, and in the same external context environment E, but based on distinct sets of interaction points, queues, and locations — see 9.2) are denoted INST(M, B, E), the set of instances of M and B in the environment E.

It must to be stressed that the definition of a module instance of the class INST (M, B, E) is recursive in that it depends on the definition of classes of instances of modules defined internally in the body B. Due to Estelle's scope rules, the depth of this recursion is always finite; thus, there is no circularity of definitions.

(c) Assuming that INST(M, B, E) is defined for each body B declared for the module header M, then the class INST(M, E) is defined as:

$$\mathrm{INST}(M, E) = \bigcup_{B} \mathrm{INST}(M, B, E)$$

This is the class of all M-instances that may result from different bodies for M within the same context E. This class is used to describe the value domains for module variables and to impose restrictions on the class of possible children instances of a module instance.

(d) The body of an Estelle specification (7.2) defines the class

INST(header-identifier, specification-identifier, ST)

for an "artificial" (nonexisting) module header named by a **header-identifier** with no parameters, no interaction points, and no exported variables, and with a standard external context environment, ST, in which only information on the Pascal basic notions are provided, e.g., the keyword **INTEGER** denotes the type domain being integers, etc. (see 9.3).

9.2 Primitives and organization of the clause

Every module instance is characterized (see 5.1) by:

(a) a set of *states*

(b) a subset of *initial states*

(c) a one-to-many *next-state function*

The above notions are defined using the following primitives:

Locations

— an infinite primitive set of locations, which may be viewed as a reservoir of abstract memory units in which arbitrary values may be stored.

Queues
: — an infinite primitive set of queues of a similar nature to Locations, to store finite but unbounded lists of interactions (messages).

Values
: — the set of possible Pascal values that may be assumed by Pascal variables.

$\perp$ — the “undefined” element which is assumed to be different from any other value in the set Values.

Ids — the set of possible Pascal identifiers (syntactic class).

It is assumed below that the usual functions **head**, **tail**, and **append** on lists (or contents of queues) are known and that **nil** denotes the empty list (see 6.2 for their definitions).

To give a characterization of an instance P of an Estelle module, in terms of (a), (b) and (c) above, the following are the starting assumptions:

— P has its own (possibly infinite) set of locations Loc_P, a subset of the set Locations, and allocation functions are defined on this set for the module variables (9.4.4).

— P has its own finite set of interaction points IP_P with a queue assigned to each of them (9.4.3). The set of queues of P is denoted Q_IP_P. Like Loc_P, Q_IP_P is not shared with other instances.

In other words, for any two different module instances P1 and P2,

$$Loc_{P1} \cap Loc_{P2} = \emptyset, Q\_IP_{P1} \cap Q\_IP_{P2} = \emptyset, \text{ and } IP_{P1} \cap IP_{P2} = \emptyset.$$

With these notations and assumptions the set of states S_P of a module instance P $\in$ INST(M, B, E) is characterized in 9.4.

The set of initial states is defined in 9.6.3; and each initial state is the result of “executing” a transition of the initialization part of the body B (the set of these transitions will be denoted ITr(B)) from the, so called, preinitial state of S_P (9.4.5.2)

The next-state function of a module instance P is defined by interpreting each transition of the transition-part of the body B (the set of these transitions will be denoted by Tr(B)) as a partial, one-to-many function in S_P (9.5.1 and 9.5.2, with interpretations of particular Estelle statements in 9.6.6). Then a mechanism of choosing one of those transitions to execute at a given moment and state is defined (9.6.2, 9.6.3 and 9.6.5).

The effect of the execution of the transition in the context of other instances is described in 9.5.3, 9.5.4 and 5.3.

9.3 External context environment and channel definition

An *external context environment*, E, consists of four functions:

$$E = (E.C, E.T, E.Ch, E.PF),$$

where

$$E.C : \text{Constant-ids} \rightarrow \text{Const-values}$$

$\text{E.T} : \text{Type-ids} \rightarrow \text{Type-domains}$

$\text{E.Ch} : \text{Channel-ids} \rightarrow (\text{Role-ids} \rightarrow (\text{Interaction-ids} \rightarrow \text{POWER(Interactions)}))$

$\text{E.PF} : \text{Procedure-function-ids} \rightarrow (\text{Values}^{m} \rightarrow \text{Values}^{n}), m = 0, 1, \ldots, n = 1, 2, \ldots,$

where m and n denote the number of arguments passed (m) and the number of values returned (n).

Where it is clear from context, we sometimes omit the word "external".

NOTES

1 All domains of the above functions such as Constant-ids, Type-ids, etc., are finite, disjoint subsets of identifiers. Therefore all four functions correspond to mappings (i.e., partial functions of finite domains) on the set of identifiers as defined by Estelle syntax. The mnemonics have been chosen for the identifier categories to differentiate these functions clearly.

2 Const-values are values of constants of Pascal as defined in Annex C. The Type-domains are value domains of types of Pascal as defined in Annex C.

3 Identifiers in the domain of the function E.PF are those of required functions and procedures of Pascal as defined in Annex C. Their meaning (in a sense of a Pascal program) shall be known when the semantics of an Estelle specification is defined. Note that they are also assumed to be **demonstrably pure**, that is, without side effects, and that is why they are presented above as operating on values rather than states. Obviously, for pure procedure calls, the corresponding state transformation is trivially derived.

The following context environment is called *standard*:

$\text{ST} = (\text{ST.C}, \text{ST.T}, \text{ST.Ch}, \text{ST.PF}),$

where

(a) $\text{dom(ST.C)} = \text{dom(ST.Ch)} = \emptyset$.

(b) dom(ST.T) = {"integer","boolean","real","char"}, and ST.T defines the type domains of these identifiers in the usual, obvious way, in accordance with the requirements of Annex C, 6.4.2.2.

(c) dom(ST.PF) comprises precisely all identifiers of required functions, and ST.PF assigns to them the meanings required in Annex C.

There is a natural extension of E.T to E.T′ defined on "type expressions" built of type identifiers in dom(E.T). This means that expressions such as:

array[1..n] of T, for $T \in \text{dom(E.T)}$,

and

```
record
    a1 : T1,
    a2 : T2,
      ...
    an : Tn
end
```

for $T1, \ldots, Tn \in \text{dom(E.T)}$, etc., also have known interpretations.

Remark: We sometimes abuse notation and use E.T to represent the extension E.T′.

For convenience, a context environment E is also often treated as a function defined over the domain:

$$\text{dom}(E) = \text{dom}(E.C) \cup \text{dom}(E.T) \cup \text{dom}(E.Ch) \cup \text{dom}(E.PF)$$

with values defined by:

$$E(x) = \begin{cases} E.C(x) & \text{if } x \in \text{dom}(E.C) \\ E.T(x) & \text{if } x \in \text{dom}(E.T) \\ \text{etc.} & \cdots \end{cases}$$

9.3.1 Interactions

The set of *interactions* is defined by:

$$\text{Interactions} = \text{Ids} \times \text{Values}^k,\ k = 0, 1, \ldots,\ \text{where Ids} \times \text{Values}^0 = \text{Ids};$$

i.e., any interaction is a sequence of values preceded by an identifier or an identifier itself. The definition of concrete subsets of interactions within a channel definition is the subject of 9.3.2.

For any interaction $\langle m, v_1, \ldots, v_k \rangle$ define:

$$\text{int-id}(\langle m, v_1, \ldots, v_k \rangle) = m.$$

9.3.2 Channel definition interpretation

In view of the above definition of a context environment, the meanings of the E.C, E.T, andE.PF mappings derive from the semantics of Pascal.

E.Ch defines the channels; i.e., for each channel identifier, given a role identifier and a interaction identifier, a set of interactions is assigned.

Given a channel definition of the general form (see 7.3.4)

```
channel H(R1, R2);
     by R1 : m1(...); m2(...); ... ; mi(...);
     by R2 : n1(...); n2(...); ... ; nj(...);
     by R1, R2 : o1(...); o2(...); ... ; ok(...);
```

where each "(...)" denotes a parameter declaration part, possibly empty, and assuming that H is in the domain of the mapping E.Ch (i.e., H ∈ dom(E.Ch)), the following conditions must be satisfied:

(a) $E.Ch(H)(x)$ is defined iff $x \in \{R1, R2\}$

(b) $E.Ch(H)(R1)(y)$ is defined iff $y \in \{m1, \ldots, mi\} \cup \{o1, \ldots, ok\}$

(c) E.Ch(H)(R2)(y) is defined iff $y \in \{n1, \ldots, nj\} \cup \{o1, \ldots, ok\}$

(d) if E.Ch(H)(x)(y) is defined and $y \in$ Ids, then E.Ch(H)(x)(y) = $\{y\}$

(e) if E.Ch(H)(x)(m) is defined and $m(p^1_1, \ldots, p^1_{k1} : T1, \ldots, p^s_1, \ldots, p^s_{ks} : Ts)$ is the interaction definition within the channel H definition, then

$$T1, T2, \ldots, Ts \in dom(E.T), \text{ and}$$

$$E.Ch(H)(x)(m) = \{m\} \times E.T(T1)^{k1} \times \ldots \times E.T(Ts)^{ks} (ki \geq 1, s \geq 1).$$

9.4 Module instances

Suppose the following two definitions occur within a body of a module:

(a) module M ... end;

(b) body B for M; ... end;

This clause is devoted to defining all notions leading to the description of the set of states (see 9.4.5) of an instance P of the class INST(M, B, E), where E is an external context environment.

9.4.1 Identifier categories of a module instance

Each pair of definitions, 9.4(a) and 9.4(b), introduces important categories of identifiers, which are listed below. These categories (a) — (m) are pairwise disjoint, and, obviously, some of them may be empty:

(a) Constant-identifiers of B ($C\text{-id}_B$) is the finite set of new constant identifiers introduced within the declaration part of B ("new" means that they occur on the left-hand-side of a constant definition).

(b) Type-identifiers of B ($T\text{-id}_B$) is the finite set of new type identifiers introduced within the declaration part of B.

(c) Channel-identifiers of B ($Ch\text{-id}_B$) is the finite set of new channel identifiers introduced within the declaration part of B.

(d) Procedure-function-identifiers of B ($PF\text{-id}_B$) is the finite set of procedure and function identifiers introduced within the declaration part of B.

(e) Module-identifiers of B ($M\text{-id}_B$) is the finite set of module header identifiers introduced within the declaration part of B.

(f) Module-body-identifiers of B ($B\text{-id}_B$) is the finite set of module body identifiers introduced within the declaration part of B.

(g) Variable-identifiers of B ($V\text{-id}_B$) is the finite set of variable identifiers introduced within the declaration part of B. The special identifier STATE is assumed to be in the set if a state-definition-part is present in B. All module variables are declared separately from Pascal variables. Therefore $MV\text{-id}_B$ denotes module variable identifiers. For technical reasons it is assumed that $MV\text{-id}_B$ is a subset of $V\text{-id}_B$.

(h) State-identifiers of B (S-id$_B$) is the finite set of state identifiers introduced within the state-definition part of B.

(i) State-set-identifiers of B (Ss-id$_B$) is the finite set of state set identifiers introduced within the state-set-definition-part of B.

(j) Intern-ip-identifiers of B (Iip-id$_B$) is the finite set of the internal interaction point identifiers introduced within the declaration part of B.

(k) Extern-ip-identifiers of M (Eip-id$_M$) is the finite set of the external interaction points declared in the module header M definition.

(l) Export-variable-identifiers of M (EV-id$_M$) is the finite set of the exported variable identifiers declared in the module header M definition.

(m) Parameter-identifiers of M (P-id$_M$) is the finite set of the parameter identifiers declared in the module header M definition.

One may also distinguish, for each transition t within the initialization or transition part of B, the following local categories, not necessarily disjoint with the above (a) — (m) categories:

(a.1) Constant-identifiers of t in B (C-id$_{B,t}$) is the finite set of constant identifiers introduced locally within the transition t block definition.

(b.1) Type-identifiers of t in B (T-id$_{B,t}$) is the finite set of type identifiers introduced locally within the transition t block definition.

(d.1) Procedure-function-identifiers of t in B (PF-id$_{B,t}$) is the finite set of procedure and function identifiers introduced within the transition t block declarations.

(g.1) Variable-identifiers of t in B (V-id$_{B,t}$) is the finite set of variable identifiers introduced locally within the transition t block definition (notice that these may be only Pascal variables).

(m.1) Interaction-param-identifiers of t in B (P-id$_{B,t}$) is the finite set of the interaction parameter identifiers occurring within the transition t. (These are parameters introduced by an interaction within a **when** clause).

All of these categories may be recognized statically, and thus it is assumed that they are known at the moment the semantics is being defined.

Remark: A finite group of interaction points or module variables may be introduced by an array construct. In that case the group identifier indexed by the element(s) of the index type(s) is a separate interaction point identifier either in the set Eip-id$_M$ $\cup$ Iip-id$_B$ or a module variable in the set MV-id$_B$. The group identifier itself is not a member of this set. E.g., if

x : array[1..3] of H(R) individual queue

is a declaration within a module header M, then x[1], x[2], and x[3] are in Eip-id$_M$, but x is not in this set. If

X : array[1..2;1..2] of M

is a declaration within a **module-variable-declaration-part**, then X[1,1], X[1,2], X[2,1], and X[2,2] are in MV-id$_B$, but X is not in this set.

9.4.2 Internal extensions of context environment and recursive assumptions

The definitions (declarations) within a body B which introduce the identifier categories (a), (b) and (c) specify a local context environment in the sense of assigning values to newly introduced identifiers of specific categories as described in 9.3. The same also concerns local definitions within each transition t within a body B, except that channels cannot be defined there. These local context environments are denoted, respectively:

E_B and $E_{B,T}$

These new contexts are necessary to define and properly to use the set of states of a module instance in the class INST(M, B, E). It is necessary to assume that instance classes for module definitions textually nested within the definition of module B are already defined (recursion). These definitions are formulated in a new context environment that is a combination of E and E_B, denoted $E[E_B]$. The two environments are combined using the covering operator of two functions defined in 6.2.

The simple meaning of the above combined context $E[E_B]$ is that, if a constant (type, channel, primitive procedure, or primitive function) identifier is redefined locally then this local definition has precedence. If there are no conflicts (i.e., the domains of these functions are disjoint), then $E[E_B]$ is a simple extension of E with definitions of E_B, the union of disjoint mappings.

Now, for identifier $M1 \in$ M-id_B it is assumed that the class INST(M1, $E[E_B]$) is already defined.

9.4.3 Interaction points of a module instance and related notions

The set of *interaction points* IP_P of a module instance P is composed of two disjoint sets of *external* (EIP_P) and *internal* (IIP_P) *interaction points* (i.e., $IP_P = EIP_P \cup IIP_P$). For an instance P in INST(M,B,E), these sets are determined by Eip-id_M and Iip-id_B sets of identifiers (see 9.4.1).

To distinguish interaction points of each instance, we establish the following convention: it is assumed that the interaction points of each instance are represented by the identifiers themselves indexed by the instance name; i.e., if ip $\in$ Eip-$id_M \cup$ Iip-id_B then $ip_P \in IP_P$. (See also the Remark on arrays of interaction points in 9.4.1). This index is omitted if it is clear from the context.

Each interaction point ip $\in IP_P$ is characterized by three elements:

(a) $send_P(ip)$ and

(b) $receive_P(ip)$ are both subsets of Interactions (9.3.1) that define respectively the set of interactions that an instance can send and receive through the interaction point ip.

(c) $queue_P(ip) \in$ Queues is a queue assigned to the interaction point ip in the set of queues of the instance P.

Thus we may talk of three functions

$$send_P, receive_P : IP_P \rightarrow \text{POWER(Interactions)}$$

and

$$queue_P : IP_P \rightarrow \text{Queues}$$

determined by the interaction point declarations of the form

(a) ip: H(R) individual queue

or

(b) ip: H(R) common queue

The first two mappings are defined as follows:

$$\text{send}_P(\text{ip}) = \bigcup_x \text{E.Ch(H)(R)(x)}$$

$$\text{receive}_P(\text{ip}) = \bigcup_x \text{E.Ch(H)(opposite(R))(x)}$$

for all x for which E.Ch(H)(R)(x) and, respectively, E.Ch(H)(opposite(R))(x) are defined (see the definition of the E.Ch mapping, 9.3.1). If the channel H was introduced by the definition "channel H(R1,R2)", then opposite(R1) = R2 and opposite(R2) = R1.

The **queue**$_P$ mapping is one particular mapping from the class satisfying the following conditions:

(a) For every ip1, ip2 $\in$ IP_P if both ip1 and ip2 are declared by (b) above, then

$\text{queue}_P(\text{ip1}) = \text{queue}_P(\text{ip2})$;

i.e., they have common queue in P.

(b) For every ip $\in$ IP_P declared by (a) above, there is no ip′ $\in$ IP_P such that

$\text{ip}' \neq \text{ip}$ and $\text{queue}_P(\text{ip}') = \text{queue}_P(\text{ip})$;

i.e., ip has an individual queue in P.

Q_IP_P will denote the set of queues of P; i.e., $Q\_IP_P = \text{queue}_P(IP_P)$, the range of the function queue_P.

The notions of the input environment and output of a module instance are derived from the above definitions in 9.4.3.1 and 9.4.3.2 below.

9.4.3.1 Input environment of P

The function

$$\text{ie}_P : Q\_IP_P \rightarrow (IP_P \times IP_P \times \text{Interactions})^*$$

is called the *input environment* of P. (A* denotes the set of all finite sequences, including the empty sequence, of elements of the set A; see 6.2).

We shall use a shorthand $ie_P(q)$ to denote $ie_P(queue_P(q))$, the contents of the queue associated with the interaction point q in the input environment ie_P.

NOTE — Contents of queues are sequences of interactions labelled by a pair of interaction points. The first serves to distinguish (in the case of a common queue) the interaction points of P itself, and the second identifies the entry point of an interaction into a hierarchy of module instances. The entry point is the interaction point at the highest level in the hierarchy of module instances which is attached to the first when the interaction is sent. These labels are needed for a proper definition of Estelle primitive statements (see 9.6.6).

Let $(p1, r1, m1), \ldots, (pk, rk, mk)$ be the contents of a queue Q of the module instance P in some input environment ie_P. We say that the sequence is *proper* iff for each $i = 1, \ldots, k$, $pi \in IP_P$, and $mi \in receive_P(pi)$ and $queue_P(pi) = Q$.

The input environment ie_P is *proper* iff $ie_P(p)$ is proper, for every $p \in IP_P$.

NOTE — The above property simply states that interactions in a queue can be only those which may be received through interaction points that the queue is associated with.

For an element $\langle p1, p2, m, v_1, \ldots, v_k \rangle$ of a queue, the function *interac* is defined by

$$interac(\langle p1, p2, m, v_1, \ldots, v_k \rangle) = \langle m, v_1, \ldots, v_k \rangle.$$

9.4.3.2 Outputs of P

The sequence of labelled interactions

$$p1.m1, \ldots, pk.mk$$

(the sequence may also be **nil**) such that, for every $i = 1, \ldots, k$, $pi \in EIP_P$ and $mi \in send_P(pi)$, is called an *external output* of P.

The set of *external outputs* of P is denoted Out_P.

NOTE — Outputs are sequences of interactions labelled by the external interaction points through which they go out.

9.4.4 Locations, variable allocations and variable visibility within a module instance

Each module instance P has its own set of locations Loc_P. This is, among other things, like interaction points and queues, one of the characteristics that differentiate module instances of the same or distinct classes.

Recall that the **when** clause opens a new scope such that identifiers of parameters of interactions ($P\text{-}id_{B,t}$) may mask identifiers declared in an outer scope.

NOTE — Allocation functions denoted by *alloc* with appropriate subscripts are introduced to define mappings of identifiers into locations (Loc_P) and other functions denoted by *vis* with appropriate subscripts are introduced to define the scope of identifiers (visibility) in a given context. The covering operator (see 6.2, f[g]), in combination with the functions above, resolves the scope of individual identifiers when there are duplicates.

If $P \in INST(M, B, E)$ then the following functions are called *allocations:*

$$alloc_B : EV\text{-}id_M \cup P\text{-}id_M \cup V\text{-}id_B \rightarrow Loc_P$$

$alloc_{Pt} : P\text{-}id_{B,t} \rightarrow Loc_P$, for every $t \in Tr(B)$
$alloc_{Vt} : V\text{-}id_{B,t} \rightarrow Loc_P$, for every $t \in Tr(B) \cup ITr(B)$

(where Tr(B) denotes transitions of the transition part of B; and ITr(B), those of the initialization part of B),

where they satisfy the following conditions:

(a) all allocation functions are one-to-one

(b) the ranges of these functions are pairwise disjoint.

NOTE — (a) and (b) explain that for each occurrence of an identifier in the appropriate sets, exactly one location is assigned based on the allocations.

The set VIS_P of *visibility mappings* contains mappings, defined on finite sets of identifiers, which make it possible to distinguish the context in which a variable identifier has been used. Three particularly useful categories of visibility mappings are defined as follows:

$vis_B = alloc_B$
$vis_{Pt} = alloc_B[alloc_{Pt}] = vis_B[alloc_{Pt}]$
$vis_{Vt} = alloc_B[alloc_{Pt}[alloc_{Vt}]] = vis_{Pt}[alloc_{Vt}]$

(See 6.2 for the definition of f[g] for any two functions.)

Example:

Let t be a single transition in a module body B with

$x1, x2, x3 \in dom(alloc_B)$, and
$x1, x2, x4 \in dom(alloc_{Pt})$, and
$x2, x5 \in dom(alloc_{Vt})$.

The allocation and visibility functions are given in Figure 4. By the definitions above:

$dom(vis_B) = dom(alloc_B) = \{x1, x2, x3\}$
$dom(vis_{Pt}) = dom(vis_B) \cup dom(alloc_{Pt}) = \{x1, x2, x3, x4\}$
$dom(vis_{Vt}) = dom(vis_{Pt}) \cup dom(alloc_{Vt}) = \{x1, x2, x3, x4, x5\}$

9.4.5 States of a module instance

For every instance $P \in INST(M, B, E)$, a state $s_P \in S_P$ is a tuple:

$s = (s.ie, s.Loc, s.children, s.Att, s.Conn, s.out)$

where

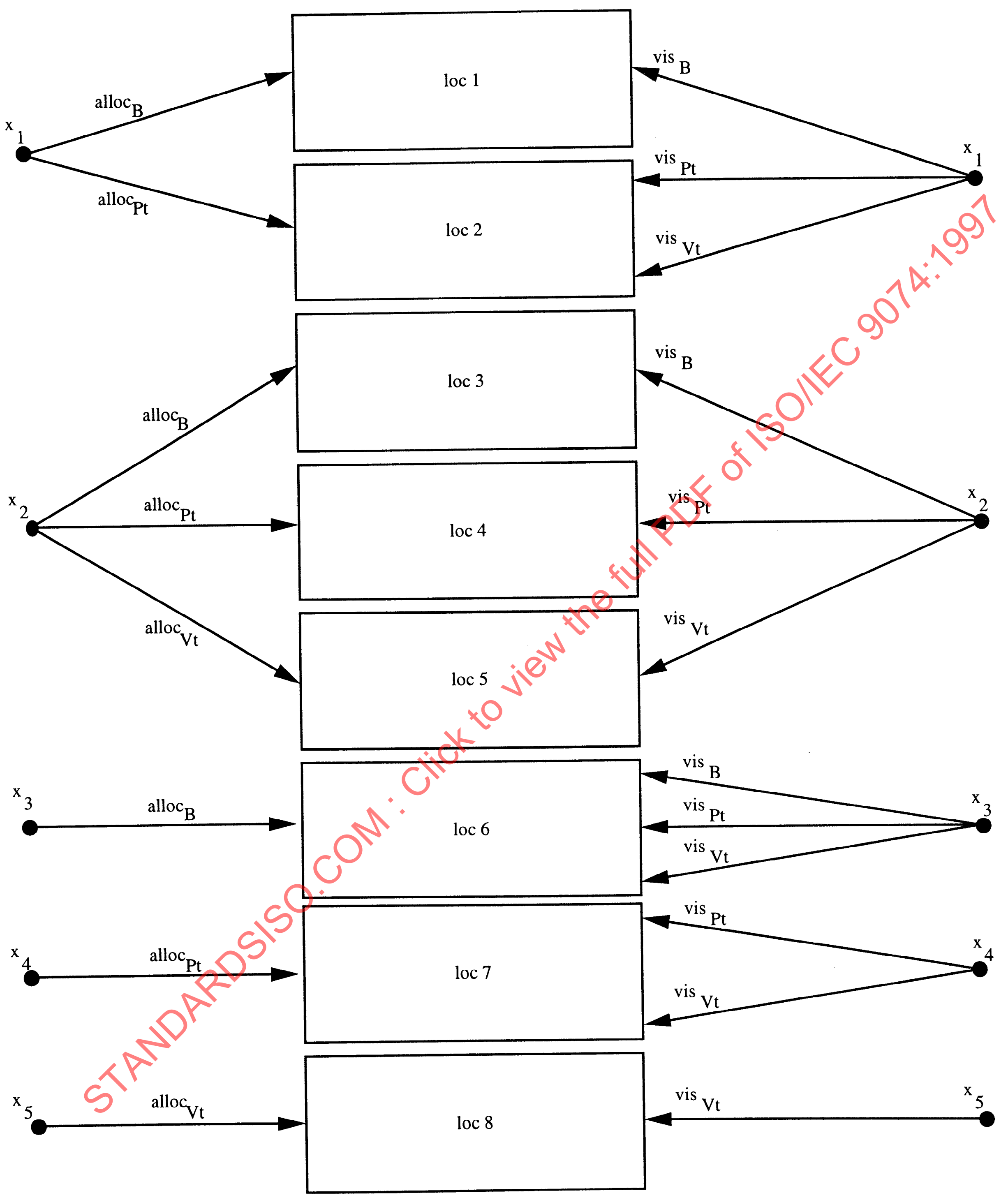

loc 1
loc 2
loc 3
loc 4
loc 5
loc 6
loc 7
loc 8
x_1
x_2
x_3
x_4
x_5
alloc_B
alloc_{Pt}
alloc_{Vt}
vis_B
vis_{Pt}
vis_{Vt}

Figure 4

(a) s.ie is an input environment of P (see 9.4.3.1), and

(b) s.Loc : $\mathrm{Loc_P} \rightarrow \mathrm{Values} \cup (\bigcup_{X \in \text{M-id}_B} \mathrm{INST}(X, E[E_B]))$

(c) s.children is a finite set of pairs (R, s_R), where $R \in \bigcup_{X \in \text{M-id}_B} \mathrm{INST}(X, E[E_B])$, and $s_R \in S_R$

(d) s.Att $\subseteq \mathrm{EIP_P} \times \bigcup_R \mathrm{EIP_R}$, for$(R, s_R) \in$ s.children

(e) s.Conn $\subseteq \mathrm{IIP_P} \times \mathrm{IIP_P} \cup (\mathrm{IIP_P} \times \bigcup_R \mathrm{EIP_R}) \cup (\bigcup_R \mathrm{EIP_R} \times \mathrm{IIP_P}) \cup (\bigcup_R \mathrm{EIP_R} \times \bigcup_R \mathrm{EIP_R})$,
for $(R, s_R) \in$ s.children

(f) s.out $\in \mathrm{Out_P}$ — an output in the state.

The index P is omitted if it does not lead to an ambiguity.

NOTES

1 Locations of P may be considered as abstract "memory units" of P in which arbitrary values as well as "references" to children instances may be stored. The mapping s.Loc gives the current contents of this memory in the state s. The contents are subject to various constraints given below and dependent on the class of instances that P belongs to.

2 s.children is intended to represent the current set of children instances of P, in the state s, where each child instance is in its own state.

3 s.Att is a binary relation between external interaction points of P and external interaction points of its children instances in the state s.

4 s.Conn is a binary relation relating either two internal interaction points of P, or an internal interaction point of P and an external interaction point of a child instance of P, or two external interaction points of children instances of P in the state s.

5 Each s.out component represents an output generated when a transition is executed; s.out grows during a transition from **nil** to include outputs as they are generated (see 9.5.4).

If $(p1, p2) \in$ s.Att, then we say that both p1 and p2 are *attached* (to each other) in the state s.

If $(p1, p2) \in$ s.Conn, then we say that both p1 and p2 are *connected* (to each other) in the state s.

If p1 occurs in s.Att or s.Conn, then p1 is *bound* in s.

9.4.5.1 Well-formedness conditions for a state

A state s is said to be well-formed if and only if each of the following conditions is satisfied:

(a) s.ie is a proper input environment as defined in 9.4.3.1.

(b) if $e \in \text{EV-id}_M$ and is declared to have a type T in the module header definition of M, then

$$\text{s.Loc}(\mathrm{alloc_B}(e)) \in E(T) \cup \{\perp\};$$

(c) if $p \in$ P-id$_M$ and is declared to have a type T in the module header definition of M, then

$$s.Loc(alloc_B(p)) \in E(T) \cup \{\perp\};$$

(d) if $x \in$ V-id$_B$ and is a Pascal variable declared to have type T within the declaration part of the body B, then

$$s.Loc(alloc_B(x)) \in E[E_B](T) \cup \{\perp\}.$$

Remark: Note the difference of environment with respect to previous definitions.

(e) If $x \in$ MV-id$_B$ and is declared to have a type $M1 \in$ M-id$_B$, then

$$s.Loc(alloc_B(x)) \in INST(M1, E[E_B]) \cup \{\perp\};$$

(f) if x = STATE and the set of state identifiers (see 7.3.10) is declared to be $(id_1, \ldots, id_n)$ then

$$s.Loc(alloc_B(x)) \in \{id_1, \ldots, id_n\};$$

(g) if $x \in$ V-id$_{B,t}$ and is declared to have type T, within the declaration internal to transition t, then

$$s.Loc(alloc_{Vt}(x)) \in E[E_B[E_{B,t}]](T) \cup \{\perp\}.$$

(h) If $p \in$ P-id$_{B,t}$, then it follows that a clause "when ip.m" occurs in the clauses of the transition t, and p is a parameter of the interaction m declared within a channel definition. Suppose that p is declared to have a type T. Then,

(1) if ip $\in$ Eip-id$_M$, then $s.Loc(alloc_{Pt}(p)) \in E(T) \cup \{\perp\}$;
(2) if ip $\in$ Iip-id$_B$, then $s.Loc(alloc_{Pt}(p)) \in E[E_B](T) \cup \{\perp\}$.

(i) If $(p1, p2) \in$ s.Att then both must refer to the same channel H and the same role R.

(j) For every $(p1, p2), (p3, p4) \in$ s.Att, $p1 = p3$ iff $p2 = p4$.

(k) If $(p1, p2) \in$ s.Conn, then both must refer to the same channel H and be of opposite roles.

(l) For every $(p1, p2), (p3, p4) \in$ s.Conn, $p1 = p3$ iff $p2 = p4$.

(m) For every pair $(p1, p2) \in$ s.Conn, $p1 \neq p2$.

(n) For every $(p1, p2) \in$ s.Conn, $(p2, p1) \in$ s.Conn.

(o) For every interaction point p, if p occurs in s.Att then p does not occur in s.Conn, and if p occurs in s.Conn then p does not occur in s.Att.

NOTES

1 Point (a) assures the correct contents of the module instance's queues in the state s.

2 Points (b) — (h) formulate restrictions on values a location can assume as determined by the type of the variable identifier associated with the location by the allocation functions.

3 Points (i) and (k) define conditions on two interaction points which may be attached or connected, respectively.

4 Points (j) and (l) say that, in a state s, each interaction point may be attached to, or connected with, at most one interaction point.

5 Point (m) says that an interaction point cannot be connected with itself, i.e., the relation s.Conn is anti-reflexive.

6 Point (n) says that the s.Conn relation is symmetric.

7 Point (o) means that an interaction point, in a state, may be attached or connected but not both.

For every interaction point ip and state $s \in S_P$, partial functions *attachment* and *connection* are defined as follows:

$$\mathrm{attachment}_P(\mathrm{ip})(s) = \mathrm{ip}' \text{ iff } \langle \mathrm{ip}, \mathrm{ip}' \rangle \in s.\mathrm{Att} \text{ or } \langle \mathrm{ip}', \mathrm{ip} \rangle \in s.\mathrm{Att}$$
$$\mathrm{connection}_P(\mathrm{ip})(s) = \mathrm{ip}' \text{ iff } \langle \mathrm{ip}, \mathrm{ip}' \rangle \in s.\mathrm{Conn}.$$

The above notions and functions will be useful in 9.6.6.

9.4.5.2 Preinitial state

A state s of a module instance P is called *preinitial* iff

$s.\mathrm{ie}(p) =$ **nil** for all $p \in \mathrm{IP}_P$ (i.e., all queues are empty)

$s.\mathrm{Loc}(k) = \bot$, for every $k \in \mathrm{Loc}_P$

$s.\mathrm{children} = \emptyset$

$s.\mathrm{Att} = \emptyset$

$s.\mathrm{Conn} = \emptyset$

$s.\mathrm{out} =$**nil** .

Denote this state $s_\bot$ (we omit the index P).

Notation:

Let

$$s(\mathrm{loc}_1/v_1, \ldots, \mathrm{loc}_n/v_n) = (s.\mathrm{ie}, s.\mathrm{Loc}(\mathrm{loc}_1/v_1, \ldots, \mathrm{loc}_n/v_n), s.\mathrm{children}, s.\mathrm{Att}, s.\mathrm{Conn}, s.\mathrm{out}).$$

This denotes the state in which values of locations $\mathrm{loc}_1, \ldots, \mathrm{loc}_n$ in s are replaced by $v_1, \ldots, v_n$, respectively, and the values of other locations remained unchanged (see 6.2 for the general definition of "/").

9.5 Transitions of a module instance

Transitions of each instance $P \in \mathrm{INST}(M, B, E)$ are simple transitions (see 7.5.2.4 and the general assumptions of clause 9) described within the initialization and the transition part of the body declaration B, i.e., in $\mathrm{ITr}(B) \cup \mathrm{Tr}(B)$. interpreted in the state space of P. Therefore each transition in B has its "instance" in P.

9.5.1 Transition statement

Let t be a transition within the initialization or transition part of the module body B, and let stm_t be the **statement-part** of the transition t; i.e.,

$$t = \text{ trans } C_t D_t \text{ begin } \text{stm}_t \text{ end },$$

where C_t and D_t denote clauses and local declarations of t, respectively, and stm_t is composed of a set of statements executed sequentially

$$\text{stm}_t = \text{stm}_1;\ \text{stm}_2;\ \ldots;\ \text{stm}_n.$$

Recall that VIS_P denotes the set of visibility mappings of variables for a given set of locations Loc_P of an instance P. The statement stm_t is then interpreted, for each module instance $P \in \text{INST}(M, B, E)$, as a partial function

$$[\text{stm}_t]_P : S_P \times \text{VIS}_P \rightsquigarrow \text{POWER}(S_P \times \text{VIS}_P)$$

with the following property:

$$(*)\ \text{if}(s', \text{vis}') \in [\text{stm}_t]_P(s, \text{vis}),\ \text{ then vis} = \text{vis}'.$$

In other words $[\text{stm}_t]_P$ is a partial function which, for an instance's state and a certain visibility of the statement variables in this state, returns sets of the instance states and does not change the visibility. By the nondeterminism caused by Estelle statements, see Remark 1 below, this is in general not a single state.

In addition to the property (*) above, the definitions within this International Standard often do not depend on the visibility argument. For these reasons and for the sake of simplicity, the following conventions are adopted:

(a) The visibility argument is omitted, where possible, from the definitions, but each time an accompanying statement will indicate the visibility mapping with respect to which a given definition has to be understood.

(b) In places where the visibility argument appears, the following notations are uniformly used:

(1) vis is used for an arbitrary visibility mapping in VIS_P with its domain including all appropriate identifiers occurring in the construct being defined:
$\text{EV-id}_M \cup \text{P-id}_M \cup \text{V-id}_B \cup \text{P-id}_{B,t} \cup \text{V-id}_{B,t}$

(2) vis_B, vis_{Pt} and vis_{Vt} are used for special categories of visibility mappings as defined in 9.4.4.

(c) In the definitions of the **exist** expression, the **forone** statement, and the **all** statement, the explicit form $\langle$state, visibility$\rangle$ is used, because in these definitions a local change to a visibility mapping must be explicitly defined.

Thus we write $[\text{stm}]_P(s)$ instead of $[\text{stm}]_P(s, \text{vis})$. The text enclosed in brackets [ ] denotes the interpretation (i.e., semantics) of the enclosed text (see 6.2).

Remark 1: The transition result $[\text{stm}_t]_P(s)$ may be composed of more than one state only if an **init**, **all**, or **forone** statement is present in stm_t. All other constructs are deterministic. Thus, in case the result reduces to a one element set $\{s'\}$, it is identified with the state s' itself.

Remark 2: The interpretations of purely Pascal constructs are assumed to be known in the above sense (obviously they are partial functions from $S_P \times VIS_P$ to $S_P \times VIS_P$). Thus, only Estelle extensions are defined.

To understand properly compound statements, the statement composition has to be interpreted as follows:

$$[\mathrm{stm1};\ \mathrm{stm2}]_P(s) = \{s' : (\text{ there exists an } s'')\ s'' \in [\mathrm{stm1}]_P(s) \text{ and } s' \in [\mathrm{stm2}]_P(s'')\}.$$

Remark 3: The composition reduces to composition of ordinary functions if the results of both statements are one element sets. The above interpretation of ";" is also assumed in the meta definitions.

9.5.2 Transition interpretation

For the rest of 9.5, it is assumed that the meaning of a statement stm_t of a transition t is already defined. The interpretation of a transition t for any module instance $P \in \mathrm{INST}(M, B, E)$ is based on the interpretation of the transition statement stm_t and two auxiliary statements that are not available in Estelle itself: *nextcontrolstate* and *reception*. These are defined as follows (for every visibility vis):

— For every $id \in Id \cup \{same\}$ and $s \in S_P$,

```
[nextcontrolstate-id]$_P$(s) =
    **if** id = same **then** s
    **else** s.Loc(alloc$_B$(state)) := id
```

— For every $ip \in IP_P$ and $s \in S_P$,

```
[reception-ip]$_P$(s) =
    **if** interac(head(s.ie(ip))) $\neq$ int-id(interac(head(s.ie(ip)))) **then**
        **let** interac(head(s.ie(ip))) = $\langle m, v_1, \ldots, v_k \rangle$
            and $p_1, \ldots, p_k$ are the interaction parameters
        **in**
            s := s(alloc$_{Pt}$($p_1$)/$v_1, \ldots,$ alloc$_{Pt}$($p_k$)/$v_k$);
    s.ie(ip) := tail(s.ie(ip)).
```

NOTE — The reception function allocates actual values of the interaction parameters (if any) and removes the top element from the ip's queue.

The *interpretation* $[t]_P$ of the transition t for the module instance $P \in \mathrm{INST}(M, B, E)$ is a partial function:

$$[t]_P : S_P \times \{vis_{Vt}\} \rightsquigarrow \mathrm{POWER}(S_P \times \{vis_{Vt}\})$$

defined as follows:

(a) If t is an input transition with "when ip.m" and "to id" clauses occurring in it, where id is an identifier declared in the state-part of the declaration-part of B or the keyword **same,** then

$$[t]_P = [\text{reception-ip}; \mathrm{stm}_t; \text{nextcontrolstate-id}]_P.$$

NOTE — Thus to interpret $[t]_P$, the result of executing the statement stm_t is extended by removing the first interaction from the ip queue and setting the value associated with the special identifier "state" to id.

(b) If t is an input transition as in (a), but the "to id" clause does not occur in it, then

$$[t]_P = [\text{reception-ip}; \text{stm}_t]_P$$

(c) If t is a spontaneous transition and the "to id" clause occurs in it, then

$$[t]_P = [\text{stm}_t; \text{nextcontrolstate-id}]_P$$

(d) Otherwise (i.e., not (a), (b), nor (c))

$$[t]_P = [\text{stm}_t]_P$$

Points (a) — (d) assign an interpretation to each transition t occurring in the module body B, i.e., in the set $\text{ITr}(B) \cup \text{Tr}(B)$.

It remains to extend this local interpretation to the global situation; i.e., to define the transmission of outputs in a global instantaneous description gid_{SP} of a specification SP, a component of which is the module instance P. This is the content of the next two clauses (9.5.3 and 9.5.4).

9.5.3 Linked interaction points

Consider a finite tree of module instances, each in one of its states and satisfying the following conditions:

(a) For any two components of the tree (P_1, s_1) and (P_2, s_2), $(P_2, s_2) \in s_1$.children iff (P_2, s_2) is a child of (P_1, s_1) in the usual parent/child relationship of a tree.

(b) If (P, s) is a leaf of the tree, then s.children = $\emptyset$.

If (P, s) is a root of such a tree, then the tree is called a *global instantaneous description of P* and is denoted by gid_P.

Remark 1: It is easy to see that each state $s \in S_P$ unambiguously determines a gid_P and vice-versa; i.e., one can use the two notions interchangeably. Depending on the context, however, it is more natural to use one notion and not the other. Nevertheless, if it is important to indicate the relationship, then $\text{gid}_{P,s}$ will denote this global instantaneous description of P which is determined by $s \in S_P$.

Remark 2: We refer to the module instance together with its state at a node as a *component* of the tree gid.

For any gid_P of a module instance P, the following definitions are introduced:

— $\text{Conn}(\text{gid}_P) = \bigcup_R s_R.\text{Conn}$

— $\text{Att}(\text{gid}_P) = \bigcup_R s_R.\text{Att}$

where the union is taken for all components (R, s_R) of the gid_P.

NOTE — The above relations are binary relations on interaction points. They represent current connections and attachments, respectively, within the tree of module instances rooted in P.

— for interaction points ip, ip′ and a $\mathrm{gid_P}$:

(a) upperattach(ip, $\mathrm{gid_P}$) = ip′ iff there is a sequence $p_0, \ldots, p_j$ ($j \geq 0$) of interactions points such that:

(1) $p_0 = ip$, $p_j = ip'$

(2) $\langle p_{i+1}, p_i \rangle \in \mathrm{Att}(\mathrm{gid_P})$, for $i = 0, \ldots, j-1$, and

(3) the sequence cannot be extended, i.e., there is no p_{j+1} such that $\langle p_{j+1}, p_j \rangle \in \mathrm{Att}(\mathrm{gid_P})$

(b) downattach(ip, $\mathrm{gid_P}$) = ip′ iff there exists a sequence $p_0, \ldots, p_j$ ($j \geq 0$) of interaction points such that:

(1) $p_0 = ip, p_j = ip'$

(2) $\langle p_i, p_{i+1} \rangle \in \mathrm{Att}(\mathrm{gid_P})$, for $i = 0, \ldots, j-1$, and

(3) the sequence cannot be extended (see (a)(3)).

The sequence itself (there exists only one) will be denoted by downattach_seq(ip, $\mathrm{gid_P}$).

(c) link(ip, $\mathrm{gid_P}$) = ip′ iff there exists $\langle ip_1, ip_2 \rangle \in \mathrm{Conn}(\mathrm{gid_P})$ where

(1) upperattach(ip, $\mathrm{gid_P}$) = ip_1, and

(2) downattach(ip_2, $\mathrm{gid_P}$) = ip′.

— linked(ip, $\mathrm{gid_P}$) is a predicate which is satisfied iff link(ip, $\mathrm{gid_P}$) is defined, i.e., if the interaction point ip is linked with some other interaction point in $\mathrm{gid_P}$.

NOTE — All relations (a), (b), (c) introduced above are well-defined partial functions by the well-formedness conditions of each component state s_R of an instance R in the $\mathrm{gid_P}$.

9.5.4 Extension of the transition interpretation over gids

Let SP be a specification module in Estelle (see 7.2). If (P, s) is a component of a $\mathrm{gid_{SP}}$, and $s' \in [t]_P(s)$, then this local change of states caused by the execution of t gives a new global instantaneous description $\mathrm{gid'_{SP}}$. Thus $[t]_P$ may be considered as a function that maps gid s into sets of gid s (for simplicity the notation is not changed). But the new states may have produced some output that has to be transmitted to the target queues in this new $\mathrm{gid'_{SP}}$.

For that reason auxiliary meta-functions *send*, *received* and *transmission* are defined as follows:

Let (P, s) and (P′, s′) be components of $\mathrm{gid_{SP}}$. We occasionally also write s = P-$\mathrm{gid_{SP}}$ to indicate that s is a current state of P in $\mathrm{gid_{SP}}$.

Let:

— head(s.out) = ⟨ip, m⟩ (the first interaction of an output),

— link(ip, $\mathrm{gid_{SP}}$) = ip′ and

— upperattach(ip′, $\mathrm{gid_{SP}}$) = ip″

Define by:

— $\mathrm{sent_P}(\mathrm{gid_{SP}})$ a new gid of SP which differs from $\mathrm{gid_{SP}}$ in that s.out := tail(s.out) (i.e., the first message from the list s.out has been sent)

— $\text{received}_P(\text{gid}_{SP})$ a new gid of SP where the difference with gid_{SP} is expressed by

$$s'.\text{ie}(ip') := \text{append}(\langle ip', ip'', m\rangle, s'.\text{ie}(ip'))$$

(i.e., the message has been received in the queue of ip′ and it has been indicated that the message entered (by a connection) through the interaction point ip″ currently attached by a sequence of attachments down to ip′.)

— $\text{transmission}_P(\text{gid}_{SP})$ =

```
let s = P-$\text{gid}_{SP}$
    in
        if s.out = nil then $\text{gid}_{SP}$
        else
            let head(s.out) = $\langle ip, m\rangle$
            in
                if linked(ip, $\text{gid}_{SP}$) then
                    let
                        link(ip, $\text{gid}_{SP}$) = ip′,
                        upperattach(ip′, $\text{gid}_{SP}$) = ip″
                    in
                        $\text{transmission}_P(\text{sent}_P(\text{received}_P(\text{gid}_{SP})))$
                else
                    $\text{transmission}_P(\text{sent}_P(\text{gid}_{SP}))$.
```

(See 6.2 for an explanation of the meta-notation used above).

NOTE — The above function transmits interactions from an output one-by-one, recursively, to the destination queues. If there is no destination queue for an interaction in the output, i.e., linked(ip, gid_{SP}) is **false**, then the interaction is discarded.

Thus the global effect of execution of an atomic transition t in the module instance P is given by the composition of the local effect of t in a state of P composed with the transmission function. This composition will be denoted by t*, and its effect is defined by:

$$t^*(\text{gid}_{SP}) = \{\text{transmission}_P(\text{gid}'_{SP}) : \text{gid}'_{SP} \in [t]_P(\text{gid}_{SP})\}$$

9.6 Interpretation of specific Estelle constructs

As in the case of statements, it is assumed that, given a state s and visibility vis, the interpretation (valuation) of a Pascal expression E, $\text{val}_P(E)(s)$, is given. For example, if x and y are variables, then

$$\text{val}_P(x + y)(s) = s.\text{Loc}(\text{vis}(x)) + s.\text{Loc}(\text{vis}(y))$$

(recall the convention, formulated in 9.5.1, on the use of the visibility argument in definitions).

9.6.1 External references

In this clause, we discuss objects that are thought of as external to a module instance.

9.6.1.1 Module-variable and interaction-point-reference

Let X be a **module-variable** or an **interaction-point-reference**; i.e., either

$X \in \text{MV-id}_B \cup \text{Eip-id}_M \cup \text{Iip-id}_B$, or

$X = Y[E_1, \ldots, E_n],\ n \geq 1,$ and E_i is an index expression and Y an identifier.

Then, for every $s \in S_P$, and visibility vis, the function *ref* is defined as follows:

ref(X)(s) = X in the first case,

and, in the second case,

$\text{ref}(X)(s) = Y[\text{val}_P(E_1)(s), \ldots, \text{val}_P(E_n)(s)]$ if the obtained value of ref(X)(s) is in $\text{MV-id}_B \cup \text{Eip-id}_M \cup \text{Iip-id}_B$,

and

ref(X)(s) is not defined, in other cases.

With the above definition, for every pair of module variable references X and Y, the interpretation of the assignment statement "X := Y" is

$\text{val}_P(X)(s) = s.\text{Loc}(\text{vis}(\text{ref}(X)(s)))$.

For $s \in S_P$ and a visibility vis,

$[X := Y]_P(s) = s(\text{loc}_x / v_y)$,

where

$\text{loc}_x = \text{vis}(\text{ref}(X)(s))$

$v_y = \text{val}_P(Y)(s)$.

If $\text{ref}(X)(s) \in \text{Eip-id}_M$, then X is called an *external ip-reference* of P. If $\text{ref}(X)(s) \in \text{Iip-id}_B$, then X is called an *internal ip-reference* of P. According to the convention in 9.4.3, if X is an interaction point identifier, then $(\text{ref}(X)(s))_P$ is the interaction point of P referenced by X in s.

9.6.1.2 Exported variables and interaction points

Let X.id occur in a transition t, where X identifies a module variable and id identifies either an exported variable or an interaction point. We introduce a function named *acs* to gain access (indirectly through a module variable) to either an exported variable or interaction point. Let $s \in S_P$, and let vis be an arbitrary visibility.

Given:

(a) $M1 \in \text{M-id}_B, B1 \in \text{B-id}_B$

(b) $\text{val}_P(\text{ref}(X)(s))(s) = R \in \text{INST}(M1, B1, E[E_B])$

(c) $(R, s_1) \in$ s.children

(d) $\text{id} \in \text{EV-id}_{M1}$

(e) $\text{ref(id)}(s1) \in \text{Eip-id}_{M1}$ with respect to the visibility vis_{B1}.

If (a) — (d) are satisfied (i.e., id is an exported variable identifier of M1), then both $\text{acs}_P(X.\text{id})(s)$ and $\text{val}_P(X.\text{id})(s)$ are defined, and

$$\text{acs}_P(X.\text{id})(s) = \text{alloc}_{B1}(\text{id})$$
$$\text{val}_P(X.\text{id})(s) = s1.\text{Loc}(\text{alloc}_{B1}(\text{id}))$$

If E is an expression, and $s \in S_P$ and vis a visibility

$$[X.\text{id} := E]_P(s) = s(\text{acs}_P(X.\text{id})(s)/\text{val}_P(E)(s)).$$

If (a) — (c) and (e) are satisfied (i.e., id is an an external interaction point reference of M1), then $\text{acs}_P(X.\text{id})(s)$ is defined, and

$$\text{acs}_P(X.\text{id})(s) = (\text{ref(id)}(s1))_R \text{ where } R = \text{val}_P(X)(s).$$

Remark 1: The function acs_P applied to a child-external-ip access returns an interaction point of a child module instance R, as might be expected. A simplifying convention was made (9.4.3) that interaction points are the denoting identifiers themselves, indexed by the module instance name. Thus the definition above.

Remark 2: It should be noted that it is statically verifiable whether or not an expression X.id is properly used within a specification. It is, however, a dynamic feature whether or not the value of the X.id expression (or the access function) is defined.

9.6.2 Semantics of transition clauses

Given a state, each clause of a transition returns a value in the set Values or is undefined. The **when** clause, **provided** clause, and **from** clause return a boolean value **true** or **false**. These clauses define a set of *enabled transitions* of a module instance in a given state. The absence of all of them makes a transition always enabled. Let us call them *enabling clauses* (9.6.3).

The **priority** clause and **delay** clause select from those enabled transitions in a state a subset of *fireable transitions* (9.6.5). The **to** clause has a purely informative character giving control-like information (linked with the **from** clause) about the next value of the control state (see 9.5.2).

The only two clauses whose values depend on visibility are the **provided** clause and the **delay** clause, since they compute values of expressions in a state. Additionally, the value of a **provided** clause may depend on its relative position with respect to a **when** clause in a transition. The following example explains the dependence.

Example:

```
provided q > 1
    when ip.m (q)
```

and

```
when ip.m (q)
    provided q > 1;
```

In the second case the value of the parameter q of the interaction m is tested in the **provided** clause. In the first the value of a variable q (which happens to be denoted by the same identifier and which is totally independent of the interaction m) is tested. Thus, a **when** clause opens a new scope to which the visibility vis_{Pt} corresponds (see 7.5.6 and 9.4.4 on visibility). This difference is formally given when enabled transitions are defined (9.6.3).

The semantics of condition clauses extends the val_P function, for a module instance $P \in INST(M, B, E)$ as follows:

(a) **when** clause

```
val_P(when p.m[ ... ])(s) = true iff
    there exists ip ∈ IP_P such that
        ip = ref(p)(s) and
        int-id(interac(head(s.ie(ip)))) = m.
```

The **when** clause is said to be *satisfied* in s if this valuation function returns **true**. (See 9.3.1 for the definition of **int-id** and 9.4.3.1 for the definition of **interac**).

(b) **provided** clause

$$val_P(\text{provided } B)(s) = val_P(B)(s)\ .$$

The **provided** clause is said to be *satisfied* in s with a given visibility vis if this valuation function returns **true**.

(c) **from** clause

Let $id_1, \ldots, id_k$ ($k \geq 1$) be a state list; i.e., either $id_i \in S\text{-}id_B$ or $id_i \in Ss\text{-}id_B$ (state-id or state-set-id, see 9.4.1)

Define the set of state identifiers of id_i, set-id_i, $i = 1, \ldots, k$, as follows:

$$\text{set-}id_i \;=\; \begin{cases} \{id_i\} & \text{if } id_i \in S\text{-}id_B \\ \{id_{i1}, \ldots, id_{ij}\} & \text{if } id_i \in Ss\text{-}id_B,\ \text{and } id_i = id_{i1}, \ldots, id_{ij},\ j \geq 1. \end{cases}$$

The set of state identifiers denoted by the state list $id_1, \ldots, id_k$ is defined to be

$$\text{from-list}(1, k) = \bigcup_{i=1}^{k} \text{set-id}_i.$$

With the above notation:

$$\text{val}_P(\text{from } id_1, \ldots, id_i)(s) = \textbf{true}$$

iff

$$\text{val}_P(\text{STATE})(s) \in \text{from-list}(1, k).$$

The **from** clause is said to be *satisfied* in s if the valuation function returns **true**.

(d) **to** clause

$$\text{val}_P(\text{to id})(s) \quad = \quad \begin{cases} \text{s.Loc(vis(STATE))} & \text{if id = same} \\ \text{“id”} & \text{otherwise.} \end{cases}$$

The visibility vis_B is the one with which a **to** clause is always interpreted.

(e) **priority** clause

$$\text{val}_P(k)(s) \quad = \quad \begin{cases} E[E_B](k) & \text{if } k \in \text{Constant-ids} \\ \text{“k”} & \text{otherwise.} \end{cases}$$

Remark: If k is a constant identifier, then value is the constant assigned to it by the declaration of k. Otherwise it is the constant “k” itself.

(f) **delay** clause

$$\text{val}_P(\text{delay}(E1, E2))(s) \quad = \quad \begin{cases} \langle \text{val}_P(E1)(s), \text{val}_P(E2)(s) \rangle & \text{if E2 is an expression} \\ \langle \text{val}_P(E1)(s), \text{infinity} \rangle & \text{if E2 = *.} \end{cases}$$

The valuation of a **delay** clause in a state with a given visibility, vis, returns a pair of values of positive integer expressions E1 and E2. The role of these pairs of values is explained in 9.6.4 and 9.6.5.

9.6.3 Enabled transitions and initial states

Let t be a transition defined in the module-body B.

Consider two different cases:

(a) t is an input transition where the **when** clause is of the form "when ip.m" (or "when ip.m(p1, . . . , pn)"), and this clause precedes a **provided** clause of t, and p1, . . . , pn ($n \geq 1$) are all parameters of the interaction identified by the interaction-id m.

In this case, the transition t is *enabled* in the state s iff

(1) if a **from** clause occurs in t, then it is satisfied in s, and

(2) the **when** clause is satisfied in s, and

(3) if $\text{interac}(\text{head}(s.\text{ie}(ip))) = \langle m, v_1, \ldots, v_k \rangle$, then the **provided** clause of t is satisfied in the state $s(\text{alloc}_{Pt}(p_1)/v_1, \ldots, \text{alloc}_{Pt}(p_n)/v_n)$ with the visibility vis_{Pt}.

(b) All other transitions.

In this case, a transition t is *enabled* in the state s iff each one of the three enabling clauses (if present) is satisfied in s with the visibility vis_B.

Let M be a module-header with parameters $p1, \ldots, pk, k \geq 0$ (by convention, $k = 0$ implies that there are no parameters). Let $T1, \ldots, Tk$ be type identifiers, where Ti is the ordinal type declared for parameter pi. Let $P \in \text{INST}(M, B, E)$ and $\text{alloc}_P(pi) = \text{loc}_i$, $i = 1, \ldots, k$. A state $s \in S_P$ is *initial* iff there is a transition $t \in \text{ITr}(B)$ and values $vi \in E(Ti)$, for $i = 1, \ldots, k$, such that

t is enabled in the state $s_\perp(\text{loc}_1/v_1, \ldots, \text{loc}_k/v_k)$, and

$s \in [t]_P(s_\perp(\text{loc}_1/v_1, \ldots, \text{loc}_k/v_k))$.

(see 9.5.2 for the definition of $[t]_P$).

NOTE — A state is initial if it is described by an initialization transition starting from preinitial state of P (9.4.5.2) with the values of the module parameters already known.

Remark: No specification module has parameters (i.e., $k = 0$ above). Initial states of a specification determine initial global instantaneous descriptions of this specification (see the Remark 1 in 9.5.3). These are of special interest as they are starting points of execution (5.3.4).

9.6.4 Delay values and time interpretation in Estelle

As noted in 5.3.5, delay values are dynamically changed by a time process independent of a specification. There are, however, interdependencies between computations and these values that must be preserved. These interdependencies are formulated below. The notions such as global situation, computation and system snapshot are introduced in 5.3.4.

The function *delays* is introduced to indicate the values of *delay timers* associated with **delay** clauses as time progresses from one situation to another given that a delayed transition is enabled. The difference between the values of this function is the elapsed time for computations executed between two successive situations.

Let $sit = (gid_{SP}; A_1, \ldots, A_n)$ be a situation in a computation, and let (P, s) be a component of gid_{SP}. For each delayed transition t of P, there is associated a pair of *delay values*. This pair is denoted

delays(t, sit)

Each value of the pair is always in the set

$$\text{Delays} = \text{real}^+ \cup \{!, \text{infinity}\},$$

where real^+ denotes non-negative reals, and for every real x, $x < \text{infinity}$, and "!" is a special element not related by "<" to any other element.

NOTE

1 The special element "!" may be thought of as representing the situation where the delay clock is not running.

The first value of the pair is sometimes called the *earliest firing time* of a transition and the second value the *latest firing time* of a transition (see the note in 9.6.5).

Remark: Everywhere in this and the next subclause, if $P = \text{INST}(M, B, E)$, then $\text{val}_P(F)(s)$ is understood to be the value of the expression F in the state s and with the visibility vis_B (see 9.4.4 for the definition of the visibility vis_B).

The following conditions must be satisfied by the **delays** function:

(a) If $\text{delays}(t, \text{sit}) = \langle a1, a2 \rangle$, then:

(1) $a1 = !$ iff $a2 = !$

(2) $0 \le a1 \le a2$ if $a1 \ne !$

(3) $a1 \ne \text{infinity}$.

(b) If sit is initial and (P, s) is a component of gid_{SP}/S_j, for some $j \in \{1, \ldots, n\}$, and t is a delayed transition of P then

$$\text{delays}(t, \text{sit}) = \langle !, ! \rangle.$$

(c) For any two consecutive situations,

$$\text{sit}_1 = (\text{gid}^1; A_1^1, \ldots, A_n^1) \text{ and } \text{sit}_2 = (\text{gid}^2; A_1^2, \ldots, A_n^2)$$

in a computation:

(1) If $A_i^1 \ne \emptyset$, then for every component (P, s) of gid^1/S_i, and every delayed transition t of P,

$$\text{delays}(t, \text{sit}_1) = \langle !, ! \rangle \text{ iff } \text{delays}(t, \text{sit}_2) = \langle !, ! \rangle.$$

(2) If $A_i^1 = \emptyset$, then for every component (P, s) of gid^1/S_i, and every delayed transition t of P containing a clause of the form "delay(E1,E2)"

(i) if $\text{delays}(t, \text{sit}_1) = \langle !, ! \rangle$ and $\text{delays}(t, \text{sit}_2) \ne \langle !, ! \rangle$,
then t is enabled in s, and $t \notin A_i^2$, and
$0 \le \text{val}_P(E1)(s) \le \text{val}_P(E2)(s)$, and
$\text{delays}(t, \text{sit}_2) = \langle \text{val}_P(E1)(s), \text{val}_P(E2)(s) \rangle$

(ii) if $\text{delays}(t, \text{sit}_1) \ne \langle !, ! \rangle$ and $\text{delays}(t, \text{sit}_2) = \langle !, ! \rangle$, then
t is not enabled in s, or $t \in A_i^2$.
Note that for $A_i^2 \ne \emptyset$ in (a) and (b) above, sit_2 is an i-th system's snapshot.

(3) If $A_i^1 = \emptyset$ and $A_i^2 \neq \emptyset$ (i.e., sit_2 is a system's snapshot and $gid^1 = gid^2$, see 5.3.4), then for every component (P, s) of gid^1/S_i, and every delayed transition t of P containing a clause of the form "delay(E1, E2)"

(i) if $delays(t, sit_1) = \langle !, ! \rangle$, and t is enabled in s, and $t \notin A_i^2$, and $0 \leq val_P(E1)(s) \leq val_P(E2)(s)$, then $delays(t, sit_2) = \langle val_P(E1)(s), val_P(E2)(s) \rangle$

(ii) if t is not enabled in s, or $t \in A_i^2$, then $delays(t, sit_2) = \langle !, ! \rangle$.

(4) For any two delayed transitions t_1 and t_2 if

$$\begin{aligned} delays(t_1, sit_1) &= \langle a_1, a_2 \rangle, \\ delays(t_2, sit_1) &= \langle b_1, b_2 \rangle, \\ delays(t_1, sit_2) &= \langle a'_1, a'_2 \rangle, \\ delays(t_2, sit_2) &= \langle b'_1, b'_2 \rangle, \text{ then} \end{aligned}$$

for $a_i, a'_i, b_i, b'_i \neq !, i = 1, 2,$

(i) $a'_2 =$ infinity iff $a_2 =$ infinity

(ii) $a'_1 \leq a_1$ and $a'_2 \leq a_2$

(iii) if $a_2 \neq$ infinity, then:
(if $a'_1 > 0$, then $a_1 - a'_1 = a_2 - a'_2$) and
(if $a'_1 = 0$, then $a_2 - a'_2 \geq a_1$)

(iv) for every $i, j = 1, 2$, if $a_i \leq b_j \neq$ infinity, then:
(if $a'_i > 0$, then $a_i - a'_i = b_j - b'_j$) and
(if $a'_i = 0$, then $b_j - b'_j \geq a_i$).

NOTES

2 Point (c)(1) says that delay timers for the delayed transitions of the i-th system cannot be turned on or off when the system is executing ($A_i^1 \neq \emptyset$) or, in other words, these clocks may be turned on or off only in between ($A_i = \emptyset$) two consecutive execution phases of this system.

3 Points (c)(2)(i) and (c)(2)(ii) explain when and how a delay timer may be turned on and off respectively.

4 Point (c)(3) says that all above timer manipulations (i.e., turning on and turning off) which are necessary in the i-th system must be already done when passing to a new snapshot of the system.

5 Point (c)(4) states that between computation steps the delay values may not increase (ii), that "infinity" value may not be changed (i), and that the delay values are changing uniformly both for earliest and latest firing values of the same transition (iii) and for those values in different transitions (iv).

(d) Let sit_j, $j \geq 0$, be a situation in a computation, and let t be a delayed transition with

$$delays(t, sit_j) = (d1, d2),$$

where $d1 > 0$. Then there is a $k > j$ such that

$$delays(t, sit_k) = (d3, d4) \text{ and } d3 < d1 \text{ or } d3 = d4 = !.$$

NOTE 6 Point (d) says that time progresses.

9.6.5 Fireable transitions and offered transitions

Let P be a module instance, $s \in S_P$, and let d1 and d2 be a pair of delay functions which to every transition t assign a pair of delay values $\langle d1(t), d2(t) \rangle$ (i.e., these are values in the set Delays satisfying condition (a) of 9.6.4).

Within a computation, if a global situation $sit = (gid_{SP}; A_1, \ldots, A_n)$ directly precedes a snapshot of the system (rooted) in S_i (see 5.3.4), then the delay values for delayed transitions that are taken into account to define fireable transitions (see the definition below) of a module instance are those given by the function **delays** introduced in 9.6.4, i.e. the delay functions d1 and d2 are such that, for every delayed transition t,

$$delays(t, sit) = \langle d1(t), d2(t) \rangle.$$

A transition $t \in Tr(B)$ is *fireable* in the state $s \in S_P$ with respect to the delay functions d1 and d2 if

(a) t is enabled in s, and if t is a delayed transition, then either $d1(t) \leq 0$, or $d1(t) = d2(t) = !$ and $0 = val_P(E1)(s) \leq val_P(E2)(s)$.

(b) if priority of t equals k, then for every other transition t′ satisfying (a) above, the priority of t′ is not less than k (i.e., the priority k is the strongest).

Remark: Since a **priority** clause may contain only constant non-negative integers, each priority is assigned completely statically to each transition t in each module instance P. By default, it is assumed that the absence of a **priority** clause within a transition's conditions means the weakest priority (i.e., the largest value) among the transition priorities of the module instance P (see 7.5.8).

$FIR_P(s, d1, d2)$ denotes all transitions, from those in Tr(B), fireable in the state s with respect to d1 and d2.

A transition t is *optionally fireable* in s with respect to d1 and d2 (written *o-fireable*) iff it is a delayed transition which is fireable in s with respect to d1 and d2, and either

(a) $d1(t) = 0$ and $d2(t) > 0$, or

(b) $d1(t) = d2(t) = !$ and $0 = val_P(E1)(s) < val_P(E2)(s)$.

$O\text{-}FIR_P(s, d1, d2)$ (an improper subset of $FIR_P(s, d1, d2)$) denotes the set of o-fireable transitions.

A transition t is *offered* for execution in s with respect to d1 and d2 iff

(c) $t \neq null$ and $t \in FIR_P(s, d1, d2)$, or

(d) $t = null$ and $O\text{-}FIR_P(d1, d2) = FIR_P(s, d1, d2)$.

Given delay functions d1 and d2, a pair (s, t) is a *local situation* of P (see 5.3.1) iff $s \in S_P$ and t is offered for execution in s with respect to d1 and d2. Remember that by the nondeterminism of Estelle, there may be more than one transition fireable in s, and hence there may be more than one local situation for a given s.

NOTE — The above definitions say, among other things, that

— Each optionally fireable transition is fireable.

— The set of offered transitions consists of all fireable transitions (point c) and possibly the "null" transition. The presence of "null" in the set of offered transitions serves to indicate that either there are no fireable transitions in s or all of them are optionally fireable (point d). Recall that the set {null} is identified with the empty set in the definition of the set of transitions selected for execution (see 5.3.3). Therefore, if there are only optionally fireable transitions in s, then each of them and "null" (i.e., "nothing") is offered by P in s.

— A delayed-transition t with delay values $\langle d1(t), d2(t) \rangle$ cannot be offered unless it remains enabled for at least d1(t) time units because earlier it is not fireable by point (a) of the definition of fireable.

— If such a delayed and fireable, henced offered, transition has remained enabled for u time units, where $d1(t) \leq u < d2(t)$, then it is optionally fireable. However, if the transition has remained enabled for d2(t) time units, then it ceases to be optionally fireable: it is fireable, but it is not o-fireable.

— A fireable transition with a delay clause of the form "delay(0,*)" is always o-fireable.

9.6.6 Semantics of primitive Estelle statements

It is convenient to introduce auxiliary meta-functions on lists of doubly labelled elements, as contents of queues are such lists, where the labels are interaction points and the elements are interactions.

Let L be such a list, i.e., $L = \langle p_1, q_1, m_1 \rangle, \ldots, \langle p_n, q_n, m_n \rangle$, or L = **nil**, where p_i, q_i are labels. Let $r_1, \ldots, r_k$, $k \geq 1$, be a sequence of labels.

The functions **erase**$_1$, **erase**$_1^*$, **erase**$_2$, and **erase**$_2^*$ are defined as follows:

$\text{erase}_1(r_1, \ldots, r_k, L)$ is the list resulting from L by erasing all of its components with the first label equal to any one in the sequence $r_1, \ldots, r_k$.

$\text{erase}_1^*(r_1, \ldots, r_k, L)$ is the list resulting from taking L and erasing all of its components with the first label different from any one in the sequence $r_1, \ldots, r_k$.

erase_2 and erase_2^* are defined similarly to erase_1 and erase_1^*, respectively, except they are based on the second label in the list.

Let r be a label, then $\text{rename}_1(r, L)$ (respectively, $\text{rename}_2(r, L)$) is the list resulting from L obtained by changing all of its first (respectively, second) labels to r.

Let L′ be another doubly labelled list, then $L \setminus L'$ is the list resulting from L after removing each of its components that also occurs in L′.

Example:

$L = \langle p_1, q_1, m_1 \rangle, \langle p_2, q_2, m_2 \rangle, \langle p_3, q_1, m_3 \rangle, \langle p_2, q_2, m_4 \rangle$

$\text{erase}_1(p_1, p_2, L) = \langle p_3, q_1, m_3 \rangle$

$\text{erase}_1^*(p_1, p_2, L) = \langle p_1, q_1, m_1 \rangle, \langle p_2, q_2, m_2 \rangle, \langle p_2, q_2, m_4 \rangle$

$\text{rename}_2(r, \text{erase}_1(p_1, p_3, L)) = \langle p_2, r, m_2 \rangle, \langle p_2, r, m_4 \rangle$

$\text{erase}_2(r, L) = L$

$\text{erase}_2^*(r, L) = \textbf{nil}$

$L \setminus \text{erase}_1(p_1, p_2, L) = \text{erase}_1^*(p_1, p_2, L)$,

$L \setminus \text{rename}_1(r, \text{erase}_1(p_1, p_3, L)) = L$

$L \setminus L = \textbf{nil}$

Henceforth we assume that states for which the semantics of Estelle statements are defined are well-formed (see 9.4.5.1). It is easy to prove from the definitions in the following clauses that well-formedness is preserved by these statements. Recall that the semantics of these statements are defined for a module instance P in the class INST(M, B, E), and for a particular transition t, and, consequently, in all of 9.6.6, the valuation function val_P is always evaluated with the visibility vis_{Vt} (9.4.4). Thus, this argument is often omitted.

9.6.6.1 Init statement

Let t be a transition in which the statement

$$\text{init X with B1}(E_1, \ldots, E_k)$$

occurs, where $E_1, \ldots, E_k$ are expressions of types compatible with the types of corresponding parameters $p_1, \ldots, p_k$ declared in the module header M1. If there are no parameters, then the statement reduces to

$$\text{init X with B1.}$$

Recall that the class $\mathrm{INST}(M1, B1, E[E_B])$ has been already defined.

Let new_P be a primitive function which for each module header $M1 \in \text{M-id}_B$ and module body $B1 \in \text{B-id}_B$ declared for M1, returns a module instance $\mathrm{new}_P(M1, B1) \in \mathrm{INST}(M1, B1, E[E_B])$ different (in the sense of the Remark below) from any other module instance existing at the moment the init statement is executed.

Remark: Module instances P and Q are *different* (9.2) iff

$$\mathrm{IP}_P \cap \mathrm{IP}_Q = \emptyset, \mathrm{Q\_IP}_P \cap \mathrm{Q\_IP}_Q = \emptyset, \text{ and } \mathrm{Loc}_P \cap \mathrm{Loc}_Q = \emptyset.$$

NOTE — In practice this means that each implementation must ensure the properties of the above remark. This may be done, however, in many ways, and therefore it is left unspecified here. From this practical point of view, the **new** function is responsible for "creating" a separate set of locations and queues and for setting the allocation and queue functions according to the given definitions.

With the above assumptions and with an arbitrary visibility vis, the interpretation of the above init statement is defined as follows:

$s' \in [\text{ init X with B1}(E_1, \ldots, E_k)]_P(s)$ iff

each s' component except s'.children and s'.Loc remains the same as in s, and

$$s'.\mathrm{Loc}(\mathrm{alloc}_B(X)) = \mathrm{new}_P(M1, B1)$$
$$s'.\mathrm{children} = s.\mathrm{children} \cup \{(\mathrm{new}_P(M1, B1), s_1)\}$$

where s_1 is a state of the module instance $R = \mathrm{new}_P(M1, B1)$, and s_1 is defined by:

Let $v_i = \mathrm{val}_P(E_i)(s)$ with the visibility vis, $i = 1, \ldots, k$,

$\mathrm{loc}_i = \mathrm{alloc}_{B1}(p_i) \in \mathrm{Loc}_R, i = 1, \ldots, k$, and
$s_0 = s_\perp(\mathrm{loc}_1/v_1, \ldots, \mathrm{loc}_k/v_k)$, where $s_\perp$ is the preinitial state of R.

Then

$s_1 \in [t]_R(s_0)$, for some $t \in ITr(B1)$ which is enabled in s_0 with the visibility vis_{B1} if such a transition exists.

Remark: Note that if no enabled transition exists, the semantics is undefined, and thus the specification is in error.

NOTES

1 The above can be read as follows: The result of an init statement in a state s with a visibility vis is that a new child instance $R = new_P(M1, B1)$ is created in its state s_1, which is the result of applying one enabled initialization transition $t \in ITr(B1)$ to the preinitial state of R in which parameter locations were loaded with actual values of expressions computed in the state s of P, with the visibility vis.

2 Note the nondeterminism involved due to the fact that the set of initialization transitions enabled in a state is not necessarily of cardinality 1.

9.6.6.2 Attach and detach statements

9.6.6.2.1 Attach

The semantics of "attach p_1 to $X.p_2$" is defined in a state s with an arbitrary visibility vis iff the following assumptions are satisfied:

(a) p_1 is an external interaction point reference defined in s (i.e., either p_1 is an identifier and $p_1 \in Eip\text{-}id_B$ or $p_1 = X[E_1, \ldots, E_n]$ and $X[val_P(E_1)(s), \ldots, val_P(E_n)(s)] \in Eip\text{-}id_B$ — see 9.6.1.1 for the ref(p1)(s) function).

(b) $acs_P(X.p_2)(s)$ is defined (see 9.6.1.2).

With these assumptions, the following notations are introduced for well-defined objects:

(a) $ip_1 = (ref(p1)(s))_P \in EIP_P$ (9.4.3 and 9.6.1.1)

(b) $ip = acs_P(X.p_2)(s)$ (9.6.1.2)

(c) $ip' = downattach(ip, gid_{P,s})$ (see 9.5.3 for both downattach and $gid_{P,s}$)

(d) $ip' \in EIP_R$ and R is a component module instance of $gid_{P,s}$

(e) $s_1 = R\text{-}gid_{P,s} \in S_R$ (read: s_1 is the current state of R in $gid_{P,s}$).

The interpretation of an attach statement of the above form is thus:

```
[attach p1 to X.p2]P(s) =
  if ip1 or ip is bound in s (9.4.5) then s
  else
      let  L1 = erase*1(ip1, s.ie(ip1)),
           L2 = rename1(ip', L1)
      in       (s1.ie(ip')  :=  append(L2, s1.ie(ip'));
                s.ie(ip1)   :=  erase1(ip1, s.ie(ip1));
                s.Att       :=  s.Att ∪ {⟨ip1, ip⟩}).
```

Remark: We discuss here an important issue concerning the semantics of several statements in 9.6.6. The constraints of 7.6 prevent an attach or connect statement from binding an interaction point that is already bound or an output statement from referencing an interaction point that is attached to a child. Any implementation should issue a warning if these situations occur. From the point of view of the formal semantics being expressed here, however, such an attempt is not an error; it is simply ignored. This choice is useful for technical convenience in some cases, and for correct specifications it has no significance.

9.6.6.2.2 Detach external interaction point

If assumption (a) of 9.6.6.2.1 is satisfied, then the semantics of the "detach p_1" statement is defined in the state s with the visibility vis.

With the notations (a), (c) — (e) of 9.6.6.2.1, the interpretation of a detach statement of the above form is defined thus:

```
[detach $p_1$]$_P$(s) =
    **if** $ip_1$ is attached in s (9.4.5) **then**
        **let**  $ip = attachment_P(ip_1)(s)$, (9.4.5)
             $L_1 = erase_1^*(ip', s_1.ie(ip'))$,
             $L_2 = erase_2(downattach\_seq(ip, gid_{P,s}), L_1)$, (9.5.3)
             $L_3 = rename_1(ip_1, L_2)$
        **in**      ($s.ie(ip_1)$  :=  $append(L_3, s.ie(ip_1))$;
                $s_1.ie(ip')$  :=  $s_1.ie(ip') \setminus L_2$;
                   $s.Att$  :=  $s.Att \setminus \{\langle ip_1, ip\rangle\}$)
    **else** s .
```

9.6.6.2.3 Detach child's interaction point

If assumption (b) of 9.6.6.2.1 is satisfied, then the semantics of the "detach $X.p_2$" statement is defined in the state s with the visibility vis.

With the notations (b) — (e) of 9.6.6.2.1, the interpretation of a detach statement of the above form is defined thus:

```
[detach $X.p_2$]$_P$(s) =
    **if** ip is attached in s **then**
        **let**  $ip_1 = attachment_P(ip)(s)$,
             $L_1 = erase_1^*(ip', s_1.ie(ip'))$,
             $L_2 = erase_2(downattach\_seq(ip, gid_{P,s}), L_1)$,
             $L_3 = rename_1(ip_1, L_2)$
        **in**      ($s.ie(ip_1)$  :=  $append(L_3, s.ie(ip_1))$
                $s_1.ie(ip')$  :=  $s_1.ie(ip') \setminus L_2$;
                   $s.Att$  :=  $s.Att \setminus \{\langle ip_1, ip\rangle\}$)
    **else** s .
```

With the same assumptions and notations, an auxiliary statement, "simple-detach $X.p_2$" is defined as follows:

```
[simple-detach X.$p_2$]$_P$(s) =
    **if** ip is attached in s **then**
        s.Att := s.Att $\setminus$ {⟨ip, attachment$_P$(ip)(s)⟩}
    **else** s .
```

Remark: The simple-detach above, which operates on a single interaction point, is used below to define the "simple-detach X" statement, which operates on a module instance.

9.6.6.2.4 Detach module

If X is a module variable and $val_P(X)(s)$ is defined, then the semantics of "detach X" is defined in the state s with the visibility vis.

Let $val_P(X)(s) \in INST(M1, B1, E[E_B])$.

With this notation, the interpretation of a detach statement of the above form is defined thus:

```
[detach X]$_P$(s) =
    **forall** r $\in$ Eip-id$_{M1}$ **do**
        s := [detach X.r]$_P$(s).
```

Remark: Note that with the assumption, $acs_P(X.r)(s)$ is defined for each $r \in$ Eip-id$_{M1}$, and therefore [detach X.r]$_P$(s) is defined.

With the same assumptions and notations, an auxiliary statement, "simple-detach X" is defined as follows:

```
[simple-detach X]$_P$(s) =
    **forall** r $\in$ Eip-id$_{M1}$ **do**
        s := [simple-detach X.r]$_P$(s).
```

Remark: The "simple-detach X" statement, which operates on a module instance, is used in the semantic definition of the terminate statement.

9.6.6.3 Connect and disconnect statements

9.6.6.3.1 Connect

The semantics of the "connect p to p′ " statement is defined in a state s with an arbitrary visibility vis iff

(a) p and p′ are internal interaction point references defined in s (see 9.6.1), or

(b) p is an internal interaction point reference defined in s and $p' = Y.p_2$ and $acs_P(Y.p_2)(s)$ is defined, or

(c) $p = X.p_1$ and $p' = Y.p_2$ and both $acs_P(X.p_1)(s)$ and $acs_P(Y.p_2)(s)$ are defined.

If one of the above assumptions is satisfied, then denote by ip and ip′ the interaction points in the state s of P corresponding to p and p′.

With this notation, the interpretation of a connect statement of the above form is defined thus:

```
[connect p to p′]$_P$(s) = [connect p′ to p]$_P$(s) =
        **if** ip or ip′ is bound in s (9.4.5) **then** s
        **else** s.Conn := s.Conn $\cup$ {⟨ip, ip′⟩, ⟨ip′, ip⟩}.
```

9.6.6.3.2 Disconnect interaction point

If p is an interaction point reference defined in s (9.6.1) or p = X.p_1 and acs(X.p_1)(s) is defined, then the semantics of the statement "disconnect p" is defined in the state s with the visibility vis. Denote by ip the interaction point corresponding to p.

With this notation, the interpretation of a disconnect statement of the above form is defined thus:

```
[disconnect p]$_P$(s) =
        if ip connected in s (9.4.5) then
            let ip′ = connection$_P$(ip)(s) (9.4.5)
            in
                s.Conn := s.Conn \ {⟨ip, ip′⟩, ⟨ip′, ip⟩}
        else s .
```

9.6.6.3.3 Disconnect module

If X is a module variable reference and val$_P$(X)(s) is defined, then the semantics of the statement "disconnect X" is defined in the state s with the visibility vis.

Let val$_P$(X)(s) $\in$ INST(M1, B1, E[E_B]).

With this notation, the interpretation of a disconnect statement of the above form is defined thus:

```
[disconnect X]$_P$(s) =
        forall r $\in$ Eip-id$_{M1}$ do
            s := [disconnect X.r]$_P$(s).
```

9.6.6.4 Release and terminate statements

The semantics of the statements "release X" and "terminate X" are defined in a state s with an arbitrary visibility vis iff X is a module variable reference and val$_P$(X)(s) is defined.

With this assumption, the interpretation of a release statement of the above form is defined thus:

```
[release X]$_P$(s) =
        s := [detach X; disconnect X]$_P$(s);
        let R = val$_P$(X)(s) and (R, $s_1$) $\in$ s.children
        in
            s.children := s.children \ {(R, $s_1$)};
        forall Y $\in$ MV-id$_B$ do
            if s.Loc(alloc$_B$(Y)) = R then
                s.Loc(alloc$_B$(Y)) := $\perp$ .
```

The interpretation of a terminate statement of the above form is defined thus:

```
[terminate X]$_P$(s) =
    s := [simple-detach X; disconnect X]$_P$(s);
    **let** R = val$_P$(X)(s) and (R, $s_1$) $\in$ s.children
    **in**
        s.children := s.children \ {(R, $s_1$)};
    **forall** Y $\in$ MV-id$_B$ **do**
        **if** s.Loc(alloc$_B$(Y)) = R **then**
            s.Loc(alloc$_B$(Y)) := $\perp$ .
```

9.6.6.5 Output statement

The semantics of the statement "output p.m($E_1, \ldots, E_n$)" is defined in a state s with an arbitrary visibility vis iff

(a) p is an interaction point reference defined in s (9.6.1)

(b) val$_P$(E_i)(s) is defined for each i = 1, . . . , n.

Remark: If the statement reduces to "output p.m", then (b) disappears.

Denote by ip = ref(p)(s) the interaction point of P corresponding to p.

With this notation, the interpretation of an output statement of the above form is defined thus:

```
[output p.m($E_1, \ldots, E_n$)]$_P$(s) =
    **if** ip is attached in s **then** s
    **else**
        **if** ip $\in$ EIP$_P$ **then**
            s.out := append($\langle$ip, m, val$_P$($E_1$)(s), . . . , val$_P$($E_n$)(s)$\rangle$, s.out)
        **else** (i.e., ip $\in$ IIP$_P$).
            **if** linked(ip, gid$_{P,s}$) (9.5.3) **then**
                **let**  ip$'$ = link(ip, gid$_{P,s}$), (9.5.3)
                     ip$''$ = connection$_P$(ip)(s), (9.4.5.1)
                     ip$'$ $\in$ IP$_R$ and (R, $s_1$) is a component of gid$_{P,s}$ (see Remark below)
                **in**
                     $s_1$.ie(ip$'$) := append($\langle$ip$'$, ip$''$, m, val$_P$($E_1$)(s), . . . , val$_P$($E_n$)(s)$\rangle$, s.ie(ip$'$))
            **else** s .
```

Remark: If ip$'$ is another internal interaction point of P, then ip$'$ = ip$''$, R = P, and s = s_1.

NOTE — Outputs to the internal interaction points are transmitted "immediately" to the destination queues (in the main **else** part), while outputs to external interaction points are "collected" during the execution of a transition and then transmitted to the environment at the end of the transition's execution (see 9.5.4 for the definition of this transmission). The reason for this difference is that, in the first case, the destination queue may be retrieved by the information included in the state s; and in the second, it may not.

9.6.6.6 Exist expression, and forone and all statements

The semantics of the following Estelle constructs are defined here:

— exist x:T suchthat E

— forone x:T suchthat E do stm_1 otherwise stm_2

— all x:T do stm

All of the above are defined in two cases: when T is a module type (i.e., T is a module-header-id in $M\text{-}id_B$) and when T is a Pascal finite ordinal type.

Recall that the semantics of each Estelle statement is considered with respect to the context in which it appears; i.e., each statement is interpreted in the set $S_P \times VIS$ (see 9.5.1). In the definitions below the visibility changes locally, so the explicit form $\langle$state, visibility$\rangle$ is used in these definitions. The global results, however, always preserve the initial visibility as postulated in 9.5.1.

Let $\langle s, vis\rangle$ be an arbitrary pair in the set $S_P \times VIS$.

The following definitions and notations are used in this clause:

— $newloc_P(L)$ is a primitive function that, for each finite subset L of location Loc_P, returns a fresh location in Loc_P, i.e., one which is not in L.

— loc = $newloc_P(dom(vis))$.

— vis_x is a new visibility function defined as follows:
$dom(vis_x) = dom(vis) \cup \{x\}$.

$$vis_x(y) \quad = \quad \begin{cases} vis(y) & \text{if } y \neq x \\ loc & \text{if } y = x. \end{cases}$$

— s-T denotes the following set:

(a) if T is a module type, then s-T = { R : R is a child instance of P in s and R $\in$ INST(T, E[E_B])}

(b) if T is an ordinal type, then s-T = E[E_B[$E_{B,t}$]](T).

— $S(s,T) = \{s' \in S_P$: s.Loc(loc) $\in$ s-T and all other components of s' are identical to those in s$\}$.

Remark 1: The definition of S(s, T) depends on the cases (a) and (b) above for s-T.

Remark 2: Note that if s-T = $\emptyset$, then S(s, T) = $\emptyset$. Note also that the set s-T is always finite.

9.6.6.6.1 Exist expression

The interpretation of an exist expression statement is defined thus:

val_P(exist x:T suchthat E)(s, vis) = **true** iff
there exists an $s' \in S(s,T)$ such that $val_P(E)(s', vis)$ = **true**.

9.6.6.6.2 Forone statement

Denote:

$D_P(s, T, E, x) =$ there exists an $s' \in S(s, T)$ such that $val_P(E)(s', vis_x) =$ **true**.

forone $(x, T, E, stm_1, stm_2) =$ forone x:T suchthat E do stm_1 otherwise stm_2.

With these notations, the interpretation of a forone statement is defined thus:

If $D_P(s, T, E, x)$, then $(s'', vis) \in [forone(x, T, E, stm_1, stm_2)]_P(s, vis)$ iff
there exists an $s' \in S(s, T)$ such that ($val_P(E)(s', vis_x) =$ **true** and $(s'', vis_x) \in [stm_1]_P(s', vis_x)$).

If $not(D_P(s, T, E, x))$, then $[forone\ (x, T, E, stm_1, stm_2)]_P(s, vis) = [stm_2]_P(s, vis)$.

9.6.6.6.3 All statement

Let u and vi be meta variables whose values are states and visibilities, respectively. The possible results of executing the statement "all x:T do stm" in the state s and visibility vis, i.e., the semantic value of

$[all\ x{:}T\ do\ stm]_P(s, vis)$

is given by all possible final values of variables u and vi of the following meta statement:

```
(u, vi) := (s, vis_x);
forall R ∈ s-T do
    (u.Loc(loc) := R;
    (u, vi) := [stm]_P(u, vi));
vi := vis.
```

Remark 1: The result of the statement is strongly nondeterministic. It depends on the order of choosing elements in the set s-T. In addition, the statement "stm" may give more than one result due to the possibility of nesting Estelle statements.

Remark 2: It is strongly recommended not to use the **all** statement if its result depends on the order of choosing elements in the set s-T.

Remark 3: In case the **exist** expression, **forone** statement, or **all** statement concerns ordinal types, the domain-list may contain several identifiers of different ordinal types, i.e., the domain list instead of being reduced to the x:T may have the following general form:

$$x^1_1, \ldots, x^1_{k1} : T_1; \ldots; x^n_1, \ldots, x^n_{kn} : T_n,$$

where $n \geq 1,\ ki \geq 1,\ i = 1, \ldots, n$.

The generalization of the above definitions is straightforward. One must create a new location loc^i_j for each identifier x^i_j and define the following:

— visibility vis_x by:

$$dom(vis_x) = dom(vis) \cup \{x^i_j : i = 1, \ldots, n; j = 1, \ldots, ki\}.$$

$$\mathrm{vis}_x(y) = \begin{cases} \mathrm{vis} & \text{if } y \neq x^i_j \\ \mathrm{loc}^i_j & \text{if } y = x^i_j. \end{cases}$$

— $s\text{-}T_i = E[E_B[E_{B,t}]](T_i), i = 1, \ldots, n.$

— $S(s, T_1, \ldots, T_n) = \{s' \in S_P : s'.\mathrm{Loc}(\mathrm{loc}^i_j) \in s\text{-}T_i$ and all other elements in s' are identical to those in s } .

Substitution of $S(s, T_1, \ldots, T_n)$ for $S(s, T)$, in the semantic definitions of an exist expression and forone statement, gives the required generalization in these cases.

In case of an all statement, the following meta statement (a particular use is the one in 9.6.6.6.3) gives the semantics of the general case:

```
(u, vi) := (s, vis_x);
forall R ∈ $(s\text{-}T_1)^{k1} \times \ldots \times (s\text{-}T_n)^{kn}$ do
    ( forall i ∈ {1, . . . , n} do
        forall j ∈ {1, . . . , ki} do
            u.Loc($\mathrm{loc}^i_j$) := R[i, j];
    (u, vi) := $[\mathrm{stm}]_P(u, vi)$);
vi := vis,
```

where $(s\text{-}T_i)^{ki} = s\text{-}T_i \times \ldots \times s\text{-}T_i$ (ki times), and R[i, j] denotes the j-th element in the cartesian product $(s\text{-}T_i)^{ki}$, the i-th part of the product $(s\text{-}T_1)^{k1} \times \cdots \times (s\text{-}T_n)^{kn}$.

ANNEX A
(informative)

Collected syntax

This annex comprises three clauses: collected syntax from clause 7, collected syntax from clause 8, and collected syntax from Annex C.

Syntax rules are defined in clauses 7, 8 and Annex C; the relationship between the parts is as follows. Annex C defines a subset of ISO Pascal [ISO/IEC 7185:1990] used by Estelle. Clause 8 defines extensions to the subset defined in annex C and summarizes the restrictions imposed by the subset. Clause 7 defines those grammar elements that are unique to Estelle. By convention, in clauses 7 and 8, nonterminal symbols written entirely in upper case refer to the corresponding lower case nonterminal symbol in the grammar rules found in Annex C (e.g., **IDENTIFIER**). Note that some nonterminals found in Annex C are extended by clause 8. For example, an applied occurrence of **COMPONENT-VARIABLE** in clause 7 refers to the nonterminal symbol "component-variable" as defined in Annex C and as extended in clause 8.

NOTES

1 The start symbol of the grammar is the nonterminal symbol **specification** (see 7.2.1).

2 The following nonterminal symbols do not appear on the right-hand side of any production and hence cannot be reached from the start symbol of the grammar: **pointer-type**, **simple-type**, **signed-number**, and **structured-type**.

3 The following nonterminal symbols appear only on the right-hand side of productions that cannot be reached from the start symbol of the grammar: **pointer-type-identifier**, **real-type-identifier**, **signed-real**, and **structured-type-identifier**.

A.1 Collected syntax from clause 7

```
7.6.1.1    actual-module-parameter = EXPRESSION .

7.6.1.1    actual-module-parameter-list = actual-module-parameter { "," actual-module-parameter } .

7.6.9.1    all-statement = "all" ( domain-list | module-domain )
                           "do" STATEMENT .

7.5.9.1    any-clause = "any" domain-list "do" .

7.6.4.1    attach-statement = "attach" external-ip "to" child-external-ip .
```

```
7.2.1      body-definition =  declaration-part
                              initialization-part
                              transition-declaration-part .

7.3.7.1    body-identifier =  IDENTIFIER .

7.3.4.1    channel-block =  +{ interaction-group } .

7.3.4.1    channel-definition =  channel-heading channel-block .

7.3.4.1    channel-heading =  “channel” IDENTIFIER “(” role-list “)” “;” .

7.3.4.1    channel-identifier =  IDENTIFIER .

7.4.2.1    child-external-ip =  module-variable “.” external-ip .

7.3.6.1    class =  “systemprocess” | “systemactivity” | “process” | “activity” .

7.5.2.1    clause-group =  [ provided-clause ]
                         ψ[ from-clause ]
                         ψ[ to-clause ]
                         ψ[ any-clause ]
                         ψ[ delay-clause ]
                         ψ[ when-clause ]
                         ψ[ priority-clause ] .

7.4.2.1    connect-ip =  child-external-ip | internal-ip .

7.6.3.1    connect-statement =  “connect” connect-ip “to” connect-ip .

7.3.1      declaration-part =  { declarations } .
```

```
7.3.1      declarations  =  CONSTANT-DEFINITION-PART
                           | TYPE-DEFINITION-PART
                           | channel-definition
                           | module-header-definition
                           | module-body-definition
                           | interaction-point-declaration-part
                           | module-variable-declaration-part
                           | VARIABLE-DECLARATION-PART
                           | state-definition-part
                           | state-set-definition-part
                           | PROCEDURE-AND-FUNCTION-DECLARATION-PART  .

7.2.1      default-options  =  "default" queue-discipline ";"  .

7.5.7.1    delay-clause  =  "delay" "(" ( EXPRESSION "," EXPRESSION
                                        | EXPRESSION "," "*"
                                        | EXPRESSION )
                                   ")"  .

7.6.6.1    detach-statement  =  "detach" ( external-ip | child-external-ip  | module-variable )  .

7.6.5.1    disconnect-statement  =  "disconnect" ( connect-ip  | module-variable )  .

7.5.9.1    domain-list  =  IDENTIFIER-LIST ":" ORDINAL-TYPE
                          { ";" IDENTIFIER-LIST ":" ORDINAL-TYPE }  .

7.6.11.1   exist-one  =  "exist" ( domain-list  | module-domain)
                          "suchthat" FACTOR  .

7.4.3.1    exported-variable  =  module-variable "." exported-variable-identifier  .

7.3.6.1    exported-variable-declaration  =  VARIABLE-DECLARATION  .
```

7.4.3.1 exported-variable-identifier = IDENTIFIER .

7.4.2.1 external-ip = interaction-point-reference .

7.6.10.1 forone-statement = "forone" (domain-list | module-domain)
"suchthat" BOOLEAN-EXPRESSION
"do" STATEMENT
["otherwise" STATEMENT] .

7.5.4.1 from-clause = "from" from-list .

7.5.4.1 from-element = state-identifier | state-set-identifier .

7.5.4.1 from-list = from-element { "," from-element } .

7.3.6.1 header-identifier = IDENTIFIER .

7.3.5.1.1 index-type-list = INDEX-TYPE { "," INDEX-TYPE } .

7.6.1.1 init-statement = "init" module-variable "with" body-identifier
["(" actual-module-parameter-list ")"] .

7.5.10.1 initialization-part = { "initialize" transition-group } .

7.3.4.1 interaction-argument-identifier = IDENTIFIER .

7.5.6.1 interaction-argument-list = "(" interaction-argument-identifier
{ "," interaction-argument-identifier } ")" .

```
7.3.4.1     interaction-definition =  IDENTIFIER
                                      [ “(” VALUE-PARAMETER-SPECIFICATION
                                      { “;” VALUE-PARAMETER-SPECIFICATION } “)”] “;” .

7.3.4.1     interaction-group =  “by” role-identifier [ “,” role-identifier ] “:”
                                 +{ interaction-definition } .

7.3.4.1     interaction-identifier =  IDENTIFIER .

7.3.5.1.1   interaction-point-declaration =  IDENTIFIER-LIST “:” interaction-point-type
                                            | IDENTIFIER-LIST “:” “array” “[” index-type-list “]”
                                                                  “of” interaction-point-type .

7.3.5.1.1   interaction-point-declaration-part =  “ip” +{ interaction-point-declaration “;” } .

7.3.5.1.1   interaction-point-identifier =  IDENTIFIER .

7.4.2.1     interaction-point-reference =  interaction-point-identifier
                                          [ “[” INDEX-EXPRESSION { “,” INDEX-EXPRESSION } “]” ] .

7.3.5.1.1   interaction-point-type = channel-identifier “(” role-identifier “)” [ queue-discipline ] .

7.6.8.1     interaction-reference =  interaction-point-reference “.” interaction-identifier .

7.4.2.1     internal-ip =  interaction-point-reference .

7.5.6.1     ip-index =  CONSTANT | VARIABLE-IDENTIFIER .
```

```
7.7.1      key-words =         "activity"         | "all"             | "any"
                             | "attach"           | "body"            | "by"
                             | "channel"          | "common"          | "connect"
                             | "default"          | "delay"           | "detach"
                             | "disconnect"       | "exist"           | "export"
                             | "external"         | "forone"          | "from"
                             | "individual"       | "init"            | "initialize"
                             | "ip"               | "module"          | "modvar"
                             | "name"             | "otherwise"       | "output"
                             | "primitive"        | "priority"        | "process"
                             | "provided"         | "pure"            | "queue"
                             | "release"          | "same"            | "specification"
                             | "state"            | "stateset"        | "suchthat"
                             | "systemactivity"   | "systemprocess"   | "terminate"
                             | "timescale"        | "trans"           | "when" .


7.3.7.1    module-body-definition =  "body" IDENTIFIER "for" header-identifier";"
                                     ( body-definition "end" ";"  | "external" ";" ) .


7.6.9.1    module-domain =  IDENTIFIER ":" header-identifier  .


7.3.6.1    module-header-definition =  "module" IDENTIFIER [class] [ "(" parameter-list ")" ] ";"
                                       [ "ip" +{ interaction-point-declaration ";" } ]
                                       [ "export" +{ exported-variable-declaration ";" } ]
                                       "end" ";"  .


7.4.1.1    module-variable =  module-variable-identifier
                             | module-variable-identifier "[" INDEX-EXPRESSION
                                                   { "," INDEX-EXPRESSION } "]"  .


7.3.9.1    module-variable-declaration =  IDENTIFIER-LIST ":" header-identifier
                                         | IDENTIFIER-LIST ":" "array" "[" index-type-list "]"
                                                                "of" header-identifier  .


7.3.9.1    module-variable-declaration-part =  "modvar" +{ module-variable-declaration ";" }  .
```

```
7.4.1.1    module-variable-identifier =  IDENTIFIER .

7.6.8.1    output-statement =  “output” interaction-reference [ ACTUAL-PARAMETER-LIST ] .

7.3.6.1    parameter-list =  VALUE-PARAMETER-SPECIFICATION
                             { “;” VALUE-PARAMETER-SPECIFICATION } .

7.5.8.1    priority-clause =  “priority” priority-constant .

7.5.8.1    priority-constant =  UNSIGNED-INTEGER  | CONSTANT-IDENTIFIER .

7.5.5.1    provided-clause =  “provided” ( BOOLEAN-EXPRESSION | “otherwise”) .

7.2.1      queue-discipline =  “common” “queue”  | “individual” “queue” .

7.6.2.1    release-statement =  “release” module-variable .

7.3.4.1    role-identifier =  IDENTIFIER .

7.3.4.1    role-list =  IDENTIFIER “,” IDENTIFIER .

7.2.1      specification =  “specification” IDENTIFIER [ system-class ] “;”
                            [ default-options ]
                            [ time-options ]
                            body-definition
                            “end” “.” .

7.3.10.1   state-definition-part =  “state” IDENTIFIER-LIST “;” .
```

7.3.10.1	state-identifier = IDENTIFIER .
7.3.11.1	state-set-constant = “[”state-identifier { “,” state-identifier } “]” .
7.3.11.1	state-set-definition = IDENTIFIER “=” state-set-constant .
7.3.11.1	state-set-definition-part = “stateset” +{ state-set-definition “;”} .
7.3.11.1	state-set-identifier = IDENTIFIER .
7.2.1	system-class = “systemprocess” | “systemactivity” .
7.6.2.1	terminate-statement = “terminate” module-variable .
7.2.1	time-options = “timescale” IDENTIFIER “;” .
7.5.3.1	to-clause = “to” to-element .
7.5.3.1	to-element = “same” | state-identifier .
7.5.2.1	transition-block = CONSTANT-DEFINITION-PART TYPE-DEFINITION-PART VARIABLE-DECLARATION-PART PROCEDURE-AND-FUNCTION-DECLARATION-PART [transition-name] STATEMENT-PART .
7.5.2.1	transition-declaration = “trans” transition-group .
7.5.2.1	transition-declaration-part = { transition-declaration } .

7.5.2.1 transition-group = +{ clause-group transition-block “;” } .

7.5.2.1 transition-name = “name” IDENTIFIER “:” .

7.5.6.1 when-clause = “when” when-ip-reference “.” interaction-identifier
[interaction-argument-list] .

7.5.6.1 when-ip-reference = interaction-point-identifier [“[” ip-index { “,” ip-index } “]”] .

A.2 Collected syntax from clause 8

8.2.7.1 ASSIGNMENT-STATEMENT = — | module-variable “:=” module-variable .

8.1.1 COMPONENT-VARIABLE = — | exported-variable .

8.2.3.2.1 CONSTANT-DEFINITION = — | IDENTIFIER “=” “any” TYPE-IDENTIFIER .

8.2.4.1 directive = letter { letter | digit } .

8.2.6.1 EXPRESSION = — | module-variable RELATIONAL-OPERATOR module-variable .

8.1.1 FACTOR = — | exist-one .

8.1.1 LETTER = — | “_” .

8.2.5.2 PROCEDURE-HEADING = — | “pure” “procedure” IDENTIFIER [FORMAL-PARAMETER-LIST] .

8.2.5.2 PROCEDURE-IDENTIFICATION = — | “pure” “procedure” PROCEDURE-IDENTIFIER .

```
8.1.1       REPETITIVE-STATEMENT  =  —  | all-statement  .

8.1.1       RESULT-TYPE  =  TYPE-IDENTIFIER  .

8.1.1       SIMPLE-STATEMENT  =  —  | attach-statement
                                    | connect-statement
                                    | detach-statement
                                    | disconnect-statement
                                    | init-statement
                                    | output-statement
                                    | release-statement
                                    | terminate-statement  .

8.1.1       STRING-CHARACTER  =  any-character-specified-in-ISO/IEC-646  .

8.1.1       STRUCTURED-STATEMENT  =  —  | forone-statement  .

8.2.3.3.1   TYPE-DEFINITION  =  —  | IDENTIFIER "=" ". . ."  .

8.1.1       WORD-SYMBOL  =  —  | key-words  .
```

A.3 Collected syntax from annex C

```
6.7.3       actual-parameter  =  expression  | variable-access  | procedure-identifier
                                 | function-identifier  .

6.7.3       actual-parameter-list  =  "(" actual-parameter { "," actual-parameter  } ")"  .

6.7.2.1     adding-operator  =  "+"  | "–"  | "or"  .

6.1.7       apostrophe-image  =  "''"  .
```

6.4.3.2	array-type = “array” “[” index-type { “,” index-type } “]” “of” component-type .
6.5.3.2	array-variable = variable-access .
6.8.2.2	assignment-statement = (variable-access | function-identifier) “:=” expression .
6.4.3.4	base-type = ordinal-type .
6.2.1	block = label-declaration-part constant-definition-part type-definition-part variable-declaration-part procedure-and-function-declaration-part statement-part .
6.7.2.3	Boolean-expression = expression .
6.4.3.3	case-constant = constant .
6.4.3.3	case-constant-list = case-constant { “,” case-constant } .
6.8.3.5	case-index = expression .
6.8.3.5	case-list-element = case-constant-list “:” statement .
6.8.3.5	case-statement = “case” case-index “of” case-list-element { “;” case-list-element } [“;”] “end” .
6.1.7	character-string = “ ' ” string-element { string-element } “ ' ” .
6.4.3.2	component-type = type-denoter .

6.5.3.1	component-variable = indexed-variable | field-designator .
6.8.3.2	compound-statement = "begin" statement-sequence "end" .
6.8.3.3	conditional-statement = if-statement | case-statement .
6.3	constant = [sign] (unsigned-number | constant-identifier) | character-string .
6.3	constant-definition = identifier "=" constant .
6.2.1	constant-definition-part = ["const" constant-definition ";" { constant-definition ";" }] .
6.3	constant-identifier = identifier .
6.8.3.9	control-variable = entire-variable .
6.1.1	digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
6.1.5	digit-sequence = digit { digit } .
6.4.4	domain-type = type-identifier .
6.8.3.4	else-part = "else" statement .
6.8.2.1	empty-statement = .
6.5.2	entire-variable = variable-identifier .

6.4.2.3 enumerated-type = “(” identifier-list “)” .

6.7.1 expression = simple-expression [relational-operator simple-expression] .

6.7.1 factor = variable-access | unsigned-constant | function-designator
| set-constructor | “(” expression “)” | “not” factor .

6.5.3.3 field-designator = record-variable “.” field-specifier | field-designator-identifier .

6.8.3.10 field-designator-identifier = identifier .

6.4.3.3 field-identifier = identifier .

6.4.3.3 field-list = [(fixed-part [“;” variant-part] | variant-part) [“;”]] .

6.5.3.3 field-specifier = field-identifier .

6.8.3.9 final-value = expression .

6.4.3.3 fixed-part = record-section { “;” record-section } .

6.8.3.9 for-statement = “for” control-variable “:=” initial-value (“to” | “downto”) final-value
“do” statement .

6.6.3.1 formal-parameter-list = “(” formal-parameter-section { “;” formal-parameter-section } “)” .

```
6.6.3.1   formal-parameter-section = value-parameter-specification
                                   | variable-parameter-specification
                                   | procedural-parameter-specification
                                   | functional-parameter-specification .

6.1.5     fractional-part = digit-sequence .

6.6.2     function-block = block .

6.6.2     function-declaration = function-heading ";" directive
                               | function-identification ";" function-block
                               | function-heading ";" function-block .

6.7.3     function-designator = function-identifier [ actual-parameter-list ] .

6.6.2     function-heading = "function" identifier [ formal-parameter-list ] ":" result-type .

6.6.2     function-identification = "function" function-identifier .

6.6.2     function-identifier = identifier .

6.6.3.1   functional-parameter-specification = function-heading .

6.8.2.4   goto-statement = "goto" label .

6.1.3     identifier = letter { letter | digit } .

6.4.2.3   identifier-list = identifier { "," identifier } .
```

6.8.3.4 if-statement = “if” Boolean-expression “then” statement [else-part] .

6.5.3.2 index-expression = expression .

6.4.3.2 index-type = ordinal-type .

6.5.3.2 indexed-variable = array-variable “[” index-expression, { “,” index-expression } “]” .

6.8.3.9 initial-value = expression .

6.1.6 label = digit-sequence .

6.2.1 label-declaration-part = [“label” label { “,” label } “;”] .

```
6.1.1     letter =       “a”  |  “b”  |  “c”  |  “d”  |  “e”   |  “f”  |  “g”
                     |   “h”  |  “i”  |  “j”  |  “k”  |  “l”   |  “m”  |  “n”
                     |   “o”  |  “p”  |  “q”  |  “r”  |  “s”   |  “t”  |  “u”
                     |   “v”  |  “w”  |  “x”  |  “y”  |  “z” .
```

6.7.1 member-designator = expression [“..” expression] .

6.7.2.1 multiplying-operator = “*” | “/” | “div” | “mod” | “and” .

6.4.2.1 new-ordinal-type = enumerated-type | subrange-type .

6.4.4 new-pointer-type = “↑” domain-type .

6.4.3.1 new-structured-type = [“packed”] unpacked-structured-type .

```
6.4.1     new-type = new-ordinal-type | new-structured-type | new-pointer-type .

6.4.2.1   ordinal-type = new-ordinal-type | ordinal-type-identifier .

6.4.2.1   ordinal-type-identifier = type-identifier .

6.4.4     pointer-type = new-pointer-type | pointer-type-identifier .

6.4.1     pointer-type-identifier = type-identifier .

6.6.3.1   procedural-parameter-specification = procedure-heading .

6.2.1     procedure-and-function-declaration-part = { ( procedure-declaration
                                                      | function-declaration ) ";" } .

6.6.1     procedure-block = block .

6.6.1     procedure-declaration = procedure-heading ";" directive
                                | procedure-identification ";" procedure-block
                                | procedure-heading ";" procedure-block .

6.6.1     procedure-heading = "procedure" identifier [ formal-parameter-list ] .

6.6.1     procedure-identification = "procedure" procedure-identifier .

6.6.1     procedure-identifier = identifier .

6.8.2.3   procedure-statement = procedure-identifier ( [ actual-parameter-list ] ) .
```

6.4.2.1	real-type-identifier = type-identifier .
6.4.3.3	record-section = identifier-list ":" type-denoter .
6.4.3.3	record-type = "record" field-list "end" .
6.5.3.3	record-variable = variable-access .
6.8.3.10	record-variable-list = record-variable { "," record-variable } .
6.7.2.1	relational-operator = "=" | "<>" | "<" | ">" | "<=" | ">=" | "in" .
6.8.3.7	repeat-statement = "repeat" statement-sequence "until" Boolean-expression .
6.8.3.6	repetitive-statement = repeat-statement | while-statement | for-statement .
6.6.2	result-type = simple-type-identifier | pointer-type-identifier .
6.1.5	scale-factor = [sign] digit-sequence .
6.7.1	set-constructor = "[" [member-designator { "," member-designator }] "]" .
6.4.3.4	set-type = "set" "of" base-type .
6.1.5	sign = "+" | "–" .
6.1.5	signed-integer = [sign] unsigned-integer .

```
6.1.5     signed-number = signed-integer | signed-real .

6.1.5     signed-real = [ sign ] unsigned-real .

6.7.1     simple-expression = [ sign ] term { adding-operator term } .

6.8.2.1   simple-statement = empty-statement | assignment-statement
                           | procedure-statement | goto-statement .

6.4.2.1   simple-type = ordinal-type | real-type-identifier .

6.4.1     simple-type-identifier = type-identifier .

6.1.2     special-symbol =     "+"   |  "-"   |  "*"   |  "/"   |  "="
                           |   "<"   |  ">"   |  "["   |  "]"   |  "."
                           |   ","   |  ":"   |  ";"   |  "↑"   |  "("
                           |   ")"   |  "<>"  |  "<="  |  ">="  |  ":="
                           |   ".."  |  word-symbol .

6.8.1     statement = [ label ":" ] ( simple-statement | structured-statement ) .

6.2.1     statement-part = compound-statement .

6.8.3.1   statement-sequence = statement { ";" statement } .

6.1.7     string-element = apostrophe-image | string-character .

6.8.3.1   structured-statement = compound-statement | conditional-statement
                               | repetitive-statement | with-statement .
```

6.4.3.1	structured-type = new-structured-type | structured-type-identifier .
6.4.1	structured-type-identifier = type-identifier .
6.4.2.4	subrange-type = constant ".." constant .
6.4.3.3	tag-field = identifier .
6.4.3.3	tag-type = ordinal-type-identifier .
6.7.1	term = factor { multiplying-operator factor } .
6.4.1	type-definition = identifier "=" type-denoter .
6.2.1	type-definition-part = ["type" type-definition ";" { type-definition ";" }] .
6.4.1	type-denoter = type-identifier | new-type .
6.4.1	type-identifier = identifier .
6.4.3.1	unpacked-structured-type = array-type | record-type | set-type .
6.7.1	unsigned-constant = unsigned-number | character-string | constant-identifier | "nil" .
6.1.5	unsigned-integer = digit-sequence .
6.1.5	unsigned-number = unsigned-integer | unsigned-real .

6.1.5	unsigned-real = digit-sequence "." fractional-part ["e" scale-factor] | digit-sequence "e" scale-factor .
6.6.3.1	value-parameter-specification = identifier-list ":" type-identifier .
6.5.1	variable-access = entire-variable | component-variable | identified-variable .
6.5.1	variable-declaration = identifier-list ":" type-denoter .
6.2.1	variable-declaration-part = ["var" variable-declaration ";" { variable-declaration ";" }] .
6.5.2	variable-identifier = identifier .
6.6.3.1	variable-parameter-specification = "var" identifier-list ":" type-identifier .
6.4.3.3	variant = case-constant-list ":" "(" field-list ")" .
6.4.3.3	variant-part = "case" variant-selector "of" variant { ";" variant } .
6.4.3.3	variant-selector = [tag-field ":"] tag-type .
6.8.3.8	while-statement = "while" Boolean-expression "do" statement .
6.8.3.10	with-statement = "with" record-variable-list "do" statement .

```
6.1.2    word-symbol =      "and"       | "array"   | "begin"   | "case"
                          | "const"     | "div"     | "do"      | "downto"
                          | "else"      | "end"     | "for"
                          | "function"  | "goto"    | "if"      | "in"
                          | "label"     | "mod"     | "nil"     | "not"
                          | "of"        | "or"      | "packed"  | "procedure"
                          | "record"    | "repeat"  | "set"
                          | "then"      | "to"      | "type"    | "until"
                          | "var"       | "while"   | "with" .
```

ANNEX B
(informative)

User guidelines

B.1 User data management

B.1.1 Purpose of user data management

User data management is an important part of a protocol specification:

— (N)-user data are received by the (N)-layer from the (N+1)-layer and combined with (N)-PCI to form (N)-PDUs, after possible fragmentation and/or concatenation. Then the (N)-PDUs are put into (N−1)-SDUs and passed to the (N−1)-layer.

— when (N−1)-SDUs are received they are decoded (i.e., the (N)-PCI are recognized and removed) and the (N)-SDUs are built (after possible reassembly) from the received data and passed to the (N+1)-layer.

B.1.2 Principles

This guideline describes preferred means for the manipulation of user date in Estelle in the form of two procedures encode and decode which may be used to perform mappings between Pascal record types and coded representations of such types as strings of octets.

PDU manipulations should be effected by means of two primitive polymorphic procedures called encode and decode with headings as follows:

procedure encode (var d : datatype ; r : rectype); primitive;

procedure decode (d : datatype ; var r : rectype); primitive;

The effect of these functions should be to map variables of any record type onto variables of datatype (as defined in 1.3 of Annex B). They should, in general, perform component-wise mapping of records onto coded octet strings and vice versa.

B.1.3 Encode procedure

A procedure called encode should be used to convert values of Pascal record types into values of octet string types. This procedure should be primitive. The procedure heading should be of the form:

procedure encode (var d : datatype ; r : rectype);
primitive ;

where datatype denotes a type of the form:

array [1 .. C] of 0 .. 255

and C denotes a positive integer constant,

and where rectype denotes any Pascal record-type.

The formal specification accompanying the use of this procedure should express the representation of values of type datatype in terms of the integral value represented by a sequence of octets each of which represents a number in the range 0 to 255 in binary notation.

B.1.4 Decode procedure

A procedure called decode should be used to convert values of octet string types into values of Pascal record types. This procedure should be primitive. The procedure heading should be of the form:

procedure decode (d : datatype ; var r : rectype); primitive ;

where datatype and rectype denote types as defined in B.1.3.

The formal specification accompanying the use of this procedure should express the representation of values of the type datatype in terms of the integral value represented by a sequence of octets each of which represents a number in the range 0 to 255 in binary notation.

B.1.5 Guidelines

The two above defined procedures (encode and decode) provide a means to describe any OSI protocol at an appropriate level of abstraction. However, should a less abstract (more implementation oriented) description be desirable, the specifier may use a more detailed manipulation of data type and provide for a more detailed description of encoding and decoding as illustrated below.

B.1.5.1 Constants and types

At the top level of the specification, the following declarations are included:

```
const
    maxdata   = any integer;        { the maximum size of any piece
                                      of data that a specification
                                      may handle }

type
    octet     = 0..255;             { one byte }
    len_type  = 0..maxdata;         { for data_type length }
    id_type   = 1..maxdata;         { for index in a data_type }

    data_type  = record             { this is a data-type record }
        l : len_type;               { actual length }
        d : array[id_type] of octet; { actual data }
    end;
```

NOTE — The fact that Pascal does not define how the array of "octet" is actually stored is not relevant since the specification of an OSI protocol defines only the relationship between the (N) and (N−1)-services in an abstract way. However problems may arise for physical layer protocols .

Whenever data must be declared, the type "data_type" will be used (i.e., some parameters of interactions and some variables in the body of modules will be of type "data_type").

B.1.5.2 Procedures and functions

At the top level of the specification the following functions and procedures will be defined:

```
{ return the actual length of the data variable }
function d_length ( data : data_type): len_type; primitive;

{ initialize the variable data to a null data_type; i.e., d_length will return 0 }
procedure d_null ( var data : data_type); primitive;

{ copy from_data into to_data }
procedure d_copy ( from_data : data_type;
                   var to_data : data_type); primitive;

{ prepare a data variable for holding data }
procedure d_create ( var data : data_type;
                     length : id_type); primitive;

{ return the octet at the specified offset; 1 is the first }
function d_get ( data : data_type;
                 offset : id_type) : octet; primitive;

{ put the octet at the specified offset; 1 is the first }
procedure d_put ( var data : data_type;
                  offset : id_type;
                  value : octet); primitive;

{ append the data contained in "addition" to the variable "base",
  and set "addition" to the null data_type }
procedure d_assemble ( var base : data_type;
                       var addition : data_type); primitive;

{ the data in old are fragmented as follows:
  the "len" first octets are moved to head;
  the tail ( of length "d_length(old) - len" ) remains in old }
procedure d_fragment ( var head : data_type;
                       var old : data_type;
                       len : len_type); primitive;
```

B.1.5.3 Definition of procedures and functions

The procedures and functions declared as primitive at the top of the specification are assumed to have the same effect as the following Pascal procedures and functions:

```
{ return the actual length of the data variable }
function d_length ( data : data_type): len_type;
begin
      d_length := data.l; { actual length of data_type }
end;

{ initialize the variable data to a null data_type; i.e d_length will return 0 }
procedure d_null ( var data : data_type);
var index : id_type; { index over the array of octets }
begin
      for index := 1 to maxdata do
            data.d[index] := 0;
      data.l := 0;
end;

{ copy from_data into to_data }
procedure d_copy ( from_data : data_type;
                         var to_data : data_type);
var index : id_type; { index over the array of octets }
begin
      for index := 1 to maxdata do
            to_data.d[index] := from_data.d[index];
      to_data.l := from_data.l;
end;

{ prepare a data variable for holding data }
procedure d_create ( var data : data_type;
                          length : id_type);
begin
      d_null(data);
      data.l := length;
end;

{ return the octet at the specified offset; 1 is the first }
function d_get ( data : data_type;
                    offset : id_type) : octet;
begin
      if ( offset > data.l) then
            d_get := 0 { return 0 if offset greater than length }
      else d_get := data.d[offset];
end;
```

```
{ put the octet at the specified offset; 1 is the first }
procedure d_put ( var data : data_type;
                  offset : id_type;
                  value : octet);
begin
    if ( offset <= data.l) then
        data.d[offset] := value;
end;

{ append the data contained in "addition" to the variable "base",
  and set "addition" to the null data_type }
procedure d_assemble ( var base : data_type;
                       var addition : data_type);
var tot_length : integer; { total length }
    index : len_type;
begin
    { compute total length and limit it to maxdata }
    tot_length := base.l+addition.l;
    if (tot_length > maxdata) then
        tot_length := maxdata;
    { append octets to base }
    for index := 1 to tot_length − base.l do
        base.d[index+base.l] := addition.d[index];
    { update length of base }
    base.l := tot_length;
    { set addition to the null data_type }
    d_null(addition);
end;

{ the data in old are fragmented as follows:
  the "len" first octets are moved to head;
  the tail ( of length "d_length(old) − len" ) remains in old }
procedure d_fragment ( var head : data_type;
                       var old : data_type;
                       len : len_type);
var index, length : len_type;
begin
    if ( len > old.l) then length := old.l
    else length := len; { length is at most the length of old }
    d_create(head, length); { initialize head to hold "length" octets }
    if (length > 0) { if length is zero, nothing to do } then begin
        { move length octets from old to head }
        for index := 1 to length do
            head.d[index] := old.d[index];
        { move tail of old at the beginning }
        for index := 1 to old.l−length do
            old.d[index] := old.d[index+length];
        { set to zero the tail of old }
```

```
            for index := old.l−length+1 to old.l do
                old.d[index] := 0;
            old.l := old.l−length;
        end;
end;
```

B.1.5.4 Example

The following gives a short example of user data manipulation based on the description in Estelle of the OSI transport protocol, class 0 [ISO 8073].

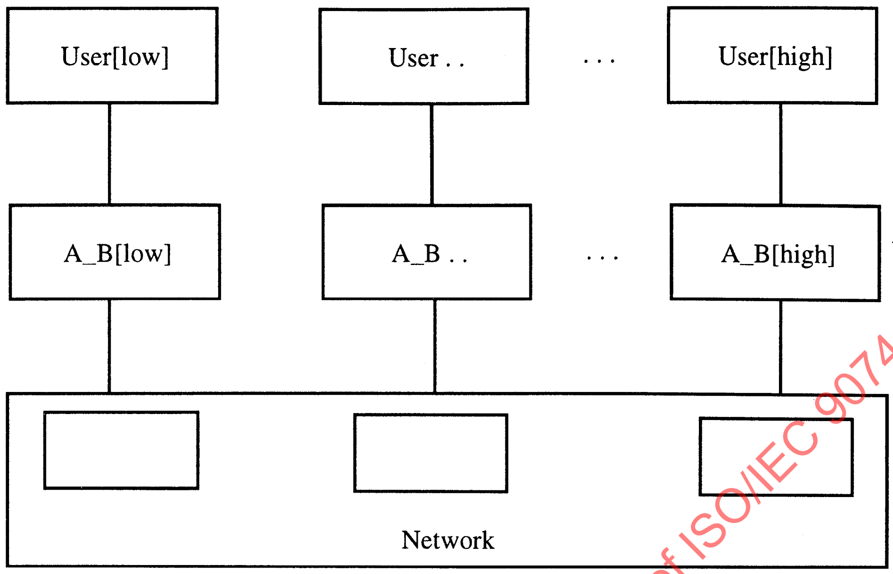
```
const
        no_eot = 0;     { not end of TSDU mark }
        eot     = 128; { end of tsdu mark }
        li_dt0  = 2;     { length indicator for data in class 0 ISO 8073 }
        cd_dt0 = 240; { pdu code for data }

var
        TDSU   : data_type; { tsdu being assembled before passing it to the user }
        pdu_len : integer;     { pdu length as negotiated during connection phase }

state
        OPEN, WFNC, WFCC, NONC; { state of transport module }

trans
        when TS.T_DT_REQ(T_user_data)
            from OPEN to SAME

            var
                nsdu,frag: data_type; { temporary data_type variable }

            procedure build_dt0 ( eot_mark : octet);
            begin
                d_create ( nsdu, 3);            { header is 3 octet in IS 8073 class 0 }
                d_put ( nsdu, 1, li_dt0);       { set the length indicator }
                d_put ( nsdu, 2, cd_dt0);       { set the PDU code }
                d_put ( nsdu, 3, eot_mark);  { set the end-of-tsdu mark }
                d_assemble ( nsdu, frag);     { append the fragment of user data }
            end;

            begin { transition block }
                while ( d_length ( T_user_data) > pdu_len − 3) do
                    begin
                        d_fragment ( frag, T_user_data, pdu_len − 3);
                        build_dt0 ( no_eot);
                        output NS.N_DT_RQ ( nsdu);
                    end;
            build_dt0 (eot);
            output NS.N_DT_RQ ( nsdu);
        end; { end transition block }


trans
    when NS.T_DT_IND(N_user_data)
        from OPEN to SAME
            provided ( { data pdu and eot }
                    ( d_get ( N_user_data, 1) = li_dt0 )
                and ( d_get ( N_user_data, 2) = cd_dt0 )
                and ( d_length ( N_user_data) <= pdu_len )
                and ( d_get ( N_user_data, 3) = eot ))

            begin { transition block }
                d_fragment ( header, N_user_data, 3); { separate the header }
                d_assemble ( TSDU , N_user_data);    { append fragment }
                output TS.T_DT_IND (TSDU);
                d_null (TSDU);
            end; { end transition block }

            provided ( { data pdu and no eot }
                    ( d_get ( N_user_data, 1) = li_dt0 )
                and ( d_get ( N_user_data, 2) = cd_dt0 )
                and ( d_length ( N_user_data) <= pdu_len )
                and ( d_get ( N_user_data,3 ) = no_eot ))

            begin { transition block }
                d_fragment ( header, N_user_data, 3); { separate the header }
                d_assemble ( TSDU , N_user_data);    { append fragment }
            end;
```

B.2 Alternating bit example

The example specification following specifies an alternating bit protocol that provides reliable communication over a network service that sometimes loses messages. This simple protocol uses a one-bit sequence number (which alternates between 0 and 1) in each message or acknowledgement to determine when messages must be retransmitted.

The specification text includes complete definitions of most module types. By using primitive functions and procedures, it omits certain implementation details.

The following diagram illustrates the global structure of the network and alternating bit entities described in the specification.



Structure of Alternating Bit System

```
Specification Example; timescale seconds;
        { This is the top level module body (specification)
          The specification has no attribute,
          and all its children ( user, ab, network ) are systems.

const { "any" base-type
           is used to specify that an implementer must
           define these constants for his environment. }

        low = any integer;              { Bounds of interaction point }
        high = any integer;             { subscripts. }
        Retran_time = any integer;      { retransmission time }
type { "..." is used to specify that an implementer
          must define these types for his environment.}
        Cep_type = low .. high;
        U_Data_type = ...;              { user data }
        Seq_type = 0 .. 1 ;             { sequence number range }
        Id_type = (DATA, ACK);
        Ndata_type =
               record
                      Id: Id_type;      { type of message }
                      Conn:Cep_type;    { cep of sender }
                      Data:U_Data_type; { user data }
                      Seq: Seq_type     { sequence number }
               end;
```

```
{ Channel definitions for communication between the processes }

channel U_access_point(User,Provider);

    by User:
        SEND_request (Udata: U_Data_type);
        RECEIVE_request;
    by Provider:
        RECEIVE_response(Udata: U_Data_type);

channel N_access_point(User,Provider);

    by User:
        DATA_request(Ndata: Ndata_type);
    by Provider:
        DATA_response(Ndata: Ndata_type);

{ Module header definitions }

module User_type systemprocess
    ( Conn_end_pt_id: Cep_type); {parameter list}
      ip U: U_access_point(User) common queue;
end; { of module header definition }

    { The interaction point is named "U"; its channel-type
      is named "U_access_point" and the role of the module
      with respect to "U" is named "User". The queue of "U"
      is shared with other interaction points designated
      "common queue" within the module named "User". }

module Alternating_bit_type systemprocess
          ( Conn_end_pt_id: Cep_type); {parameter list}
            ip {interaction point list}
              U: U_access_point(Provider) common queue;
              N: N_access_point(User) individual queue;
end; { of module header definition }

    { The module has two interaction points named "U" and "N";
      the roles of the module are named:
          "Provider" with respect to "U", and
          "User" with respect to "N".
      Notice that there is an individual queue associated
      with the interaction point "N". If the queue would
      have been common (with the queue of "U") a
      SEND_request interaction output by the user while
      the module is the in the ACK_WAIT state would lead
      to a deadlock (since the module would not be able
      to process a network interaction put in the same queue).}
```

```
module Network_type systemprocess ;
            ip { interaction point list }
                N: array[Cep_type] of
                    N_access_point(Provider) individual queue;
end; { of module header definition }

      { "N" is logically partitioned into an array
        of identical interaction points; each may be
        identified by a subscript whose type is "Cep_type".}

{ Body definitions for modules }

body Network_body for Network_type; external;

body User_body for User_type;external;

{ The body for alternating bit is defined below: }

body Alternating_bit_body for Alternating_bit_type;

type
      Msg_type =
            record { record introduces a data structure }
                  Msgdata: U_Data_type; { to be "..." }
                  Msgseq: Seq_type
            end;

      Buffer_type = ...;

var
      Send_buffer, Recv_buffer: Buffer_type;
      Send_seq, Recv_seq: Seq_type;
      P, Q: Msg_type;
      B: Ndata_type;

state ACK_WAIT, ESTAB; { state definition part }

stateset { state-set-definition-part }
  EITHER = [ACK_WAIT, ESTAB];

function Ack_ok(Nd: Ndata_type): boolean;
        {notice that a function shall be demonstrably pure }
  begin
        Ack_ok := (Nd.Id = ACK) and (Nd.Seq = Send_seq);
  end;
```

```
procedure Copy(var To_Data: U_Data_type; From_data: U_Data_type);
primitive; { procedure provided by implementer: copy a user data variable }

procedure Empty(var Data: U_Data_type);
primitive; { procedure provided by implementer:
             initialize a variable holding user data to
             the value "no user data" }

procedure Format_data(Msg: Msg_type; var B: Ndata_type);
  begin
       B.Id := DATA;
       B.Conn := Conn_end_pt_id;
       { connection reference given in the instantiation }
       copy( B.Data, Msg.Msgdata); { copy data }
       B.Seq := Msg.Msgseq;
  end;

procedure Format_ack (Msg: Msg_type; var B: Ndata_type);
  begin
       B.Id := ACK;
       B.Conn := Conn_end_pt_id;
       empty (B.Data); { no data for an ACK }
       B.Seq := Msg.Msgseq;
  end;

 { two variables of type "buffer_type" are used to hold messages ( of type "msg_type"):
   Send_buffer for sending, Receive_buffer for receiving.
   The following procedures and functions are used
   manipulate buffer_type variables }

procedure Empty_buf(var Buf: Buffer_type);
primitive; { procedure provided by implementer :
             set a buffer to "empty" i.e. contains no messages}

procedure Store(var Buf: Buffer_type; Msg: Msg_type);
primitive; { procedure provided by implementer :
             store a message into a buffer_type
             variable such that the messages can be
             retrieved or removed in a FIFO manner}

procedure Remove(var Buf: Buffer_type);
primitive; { procedure provided by implementer :
             remove the first message}

function Retrieve(Buf: Buffer_type): Msg_type;
primitive; { function provided by implementer :
             retrieve the first message and return it;
             the message is not removed }
```

```
function buffer_empty(Buf: Buffer_type) : boolean;
primitive; { function provided by implementer :
                check if a buffer contains a message}

procedure Inc_send_seq;
  begin
        Send_seq := (Send_seq + 1) mod 2
  end;

procedure Inc_recv_seq;
  begin
        Recv_seq := (Recv_seq + 1) mod 2
  end;

initialize { initialization-part of the alternating bit process }

to ESTAB { initialize major state variable to "ESTAB" }
     begin { initialize variables }
             Send_seq := 0;
             Recv_seq := 0;
             Empty_buf (Send_buffer); { implementation specific }
             Empty_buf (Recv_buffer); { implementation specific }
     end;

trans { transition-declaration-part of the alternating bit process }

from ESTAB
       to ACK_WAIT
             when U.SEND_request
              begin                            { transition 1 }
                    copy(P.Msgdata,Udata);     { copy user data in P }
                    P.Msgseq := Send_seq;      { P holds the sending seq num }
                    Store(Send_buffer,P);      { store P in sending buffer }
                    Format_data(P,B);          { format a network message }
                    output N.DATA_request(B);
              end;

from EITHER
       to same
             when U.RECEIVE_request
                   provided not buffer_empty(Recv_buffer)
                    begin                      { transition 2 }
                          Q := Retrieve(Recv_buffer);   { retrieve received message }
                          output U.RECEIVE_response(Q.Msgdata);
                          Remove(Recv_buffer)  { remove message from receiving buffer }
                    end;
```

```
from ACK_WAIT
    to ACK_WAIT
        delay (Retran_time)
         begin                              { transition 3 }
            P := Retrieve(Send_buffer);     { retrieve message to be retransmitted }
            Format_data(P,B);               { format a network message }
            output N.DATA_request(B);
         end;


from ACK_WAIT
    to ESTAB
        when N.DATA_response
            provided Ack_ok(Ndata)
                begin                       { transition 4 }
                    Remove(Send_buffer);    { remove acknowledged message }
                    Inc_send_seq
                end;

from EITHER
    to same
        when N.DATA_response
            provided Ndata.Id = DATA
                begin                       { transition 5 }
                    copy (Q.Msgdata,Ndata.Data);
                    Q.Msgseq := Ndata.Seq;
                    Format_ack(Q,B);
                    output N.DATA_request(B);
                    if Ndata.Seq = Recv_seq then
                        begin
                            Store(Recv_buffer,Q);
                            Inc_recv_seq
                        end
                end;

end; { of the Alternating_bit_body}

modvar
    { module-variable-declaration-part of the specification }

    User: array [Cep_type] of User_type;
    Alternating_bit: array[Cep_type] of Alternating_bit_type;
    Network: Network_type;
```

```
initialize { initialization-part of the specification }

    begin { module initialization }

        init Network with Network_body;

        { note how the repetitive operator "all" is used for each
          instance of a connection end point (Cep); i.e., each
          interaction point has end points at two modules. }

        all Cep: Cep_type do
            begin
                init User[Cep] with User_body(Cep);
                init Alternating_bit[Cep] with Alternating_bit_body(Cep);
                { connect interaction points }
                connect User[Cep].U to Alternating_bit[Cep].U;
                connect Alternating_bit[Cep].N to Network.N[Cep];
            end;
    end; { of module initialization within the specification's initialization-part }

end. { End of specification; the specification has no transition part }
```

ANNEX C (normative)

Pascal subset used by Estelle

The following text is a modification of [ISO/IEC 7185:1990]. Original clause numbers should help the reader note the changes.

NOTE — Symbols deleted from production rules are noted in the following text. Extensions to production rules appear in clause 8.

6 Requirements

6.1 Lexical tokens

NOTE — The syntax given in this subclause describes the formation of lexical tokens from characters and the separation of these tokens and therefore does not adhere to the same rules as the syntax in the rest of this International Standard.

6.1.1 General

The lexical tokens used to construct Pascal programs are classified into special-symbols, identifiers, directives, unsigned-numbers, labels, and character-strings. The representation of any letter (upper case or lower case, differences of font, etc.) occurring anywhere outside of a character-string (see **6.1.7**) shall be insignificant in that occurrence to the meaning of the specification.

```
letter =    "a"  | "b"  | "c"  | "d"  | "e"  | "f"  | "g"
          | "h"  | "i"  | "j"  | "k"  | "l"  | "m"  | "n"
          | "o"  | "p"  | "q"  | "r"  | "s"  | "t"  | "u"
          | "v"  | "w"  | "x"  | "y"  | "z" .
```

NOTE — Clause 8 extends the production rule for **letter**.

```
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
```

6.1.2 Special-symbols

The special-symbols are tokens having special meanings and are used to delimit the syntactic units of the language.

```
special-symbol =    "+"  | "−"           | "*"   | "/"   | "="
                  | "<"  | ">"           | "["   | "]"   | "."
                  | ","  | ":"           | ";"   | "↑"   | "("
                  | ")"  | "<>"          | "<="  | ">="  | ":="
                  | ".." | word-symbol .
```

```
word-symbol =     "and"       | "array"   | "begin"   | "case"
                | "const"     | "div"     | "do"      | "downto"
                | "else"      | "end"     | "for"
                | "function"  | "goto"    | "if"      | "in"
                | "label"     | "mod"     | "nil"     | "not"
                | "of"        | "or"      | "packed"  | "procedure"
                | "record"    | "repeat"  | "set"
                | "then"      | "to"      | "type"    | "until"
                | "var"       | "while"   | "with" .
```

NOTE — The reserved words "file" and "program" have been deleted.

6.1.3 Identifiers

Identifiers can be of any length. The *spelling* of an identifier shall be composed from all its constituent characters taken in textual order, without regard for the case of letters. No identifier shall have the same spelling as any word-symbol. Identifiers that are specified to be *required* shall have special significance (see **6.2.2.10** and **6.10**).

identifier = letter { letter | digit } .

Examples:

```
X
time
readinteger
WG4
AlterHeatSetting
InquireWorkstationTransformation
InquireWorkstationIdentification
tr_connect_req
ss_sync_major
```

NOTE — see clause 8 for **directive**.

6.1.5 Numbers

An unsigned-integer shall denote in decimal notation a value of integer-type (see **6.4.2.2**). An unsigned-real shall denote in decimal notation a value of real-type (see **6.4.2.2**). The letter "e" preceding a scale-factor shall mean *times ten to the power of.* The value denoted by an unsigned-integer shall be greater than or equal to 0 (see **6.4.2.2** and **6.7.2.2**).

signed-number = signed-integer | signed-real .

signed-real = [sign] unsigned-real .

signed-integer = [sign] unsigned-integer .

unsigned-number = unsigned-integer | unsigned-real .

sign = "+" | "−" .

unsigned-real = digit-sequence "." fractional-part ["e" scale-factor]
| digit-sequence "e" scale-factor .

unsigned-integer = digit-sequence .

fractional-part = digit-sequence .

scale-factor = [sign] digit-sequence .

digit-sequence = digit { digit } .

Examples:

```
1e10
1
+100
-0.1
5e-3
87.35E+8
```

6.1.6 Labels

Labels shall be digit-sequences and shall be distinguished by their apparent integral values and shall be in the closed interval 0 to 9999. The *spelling* of a label shall be its apparent integral value.

label = digit-sequence .

6.1.7 Character-strings

A character-string containing a single string-element shall denote a value of the required char-type (see **6.4.2.2**). A character-string containing more than one string-element shall denote a value of a string-type (see **6.4.3.2**) with the same number of components as the character-string contains string-elements. All character-strings with a given number of components shall possess the same string-type.

There shall be an implementation-defined one-to-one correspondence between the set of alternatives from which string-elements are drawn and a subset of the values of the required char-type. The occurrence of a string-element in a character-string shall denote the occurrence of the corresponding value of char-type.

character-string = " ' " string-element { string-element } " ' " .

string-element = apostrophe-image | string-character .

apostrophe-image = " '' " .

NOTE — Conventionally, the apostrophe-image is regarded as a substitute for the apostrophe character, which cannot be a string-character.

Examples:

```
'A'
';'
''''
'Pascal'
```

```
'THIS IS A STRING'
```

6.1.8 Token separators

Where a *commentary* shall be any sequence of characters and separations of lines, containing neither } nor *), the construct

("{" | "(*") commentary ("*)" | "}")

shall be a *comment* if neither the { nor the (* occurs within a character-string or within a commentary.

NOTES

1 A comment may thus commence with { and end with *), or commence with (* and end with }.

2 The sequence (*) cannot occur in a commentary even though the sequence {} can.

Comments, spaces (except in character-strings), and the separations of consecutive lines shall be considered to be token separators. Zero or more token separators can occur between any two consecutive tokens, before the first token of a specification text, or after the last token of the specification text. There shall be at least one separator between any pair of consecutive tokens made up of identifiers, word-symbols, labels or unsigned-numbers. No separators shall occur within tokens.

6.1.9 Lexical alternatives

The representation for lexical tokens and separators given in **6.1.1** to **6.1.8**, except for the character sequences (* and *), shall constitute a *reference representation* for these tokens and separators.

To facilitate the use of Pascal on processors that do not support the reference representation, the following alternatives have been defined. All processors that have the required characters in their character set shall provide both the reference representations and the alternative representations, and the corresponding tokens or separators shall not be distinguished. Provision of the reference representations, and of the alterative token @, shall be implementation-defined.

The alternative representations for the tokens shall be

Reference token	**Alternative token**
↑	@
[	(.
]	.)

NOTE — 1 The character ↑ that appears in some national variants of ISO 646 is regarded as identical to the character ^. In this International Standard, the character ↑ has been used because of its greater visibility.

The comment-delimiting characters { and } shall be the reference representations, and (* and *) respectively shall be alternative representations (see **6.1.8**).

6.2 Blocks, scopes, and activations

6.2.1 Blocks

A block closest-containing a label-declaration-part in which a label occurs shall closest-contain exactly one statement in which that label occurs. The occurrence of a label in the label-declaration-part of a block shall be its defining-point for the region that is the block. Each applied occurrence of that label (see **6.2.2.8**) shall be a label. Within an activation of the block, all applied occurrences of that label shall denote the corresponding program-point in the algorithm of the activation at that statement (see **6.2.3.2 b)**).

```
block = label-declaration-part constant-definition-part type-definition-part
        variable-declaration-part procedure-and-function-declaration-part
        statement-part .

label-declaration-part = [ "label" label { "," label } ";" ] .

constant-definition-part = [ "const" constant-definition ";" { constant-definition ";" } ] .

type-definition-part = [ "type" type-definition ";" { type-definition ";" } ] .

variable-declaration-part = [ "var" variable-declaration ";" { variable-declaration ";" } ] .

procedure-and-function-declaration-part = { ( procedure-declaration
                                            | function-declaration ) ";" } .
```

The statement-part shall specify the algorithmic actions to be executed upon an activation of the block.

```
statement-part = compound-statement .
```

6.2.2 Scopes

See 7.1.

6.2.3 Activations

6.2.3.1

A procedure-identifier or function-identifier having a defining-point for a region that is a block within the procedure-and-function-declaration-part of that block shall be designated *local* to that block.

6.2.3.2

The activation of a block shall contain

a) for the statement-part of the block, an algorithm, the completion of which shall terminate the activation (see also **6.8.2.4**);

b) for each defining-point of a label in the label-declaration-part of the block, a corresponding program-point (see **6.2.1**);

c) for each variable-identifier having a defining-point for the region that is the block, a variable possessing the type associated with the variable-identifier;

d) for each procedure-identifier local to the block, a procedure with the procedure-block corresponding to the procedure-identifier, and the formal-parameters of that procedure-block;

e) for each function-identifier local to the block, a function with the function-block corresponding to, and the result type associated with, the function-identifier, and the formal-parameters of that function-block;

f) if the block is a function-block, a result possessing the associated result type.

NOTE — Each activation contains its own algorithm, program-points, variables, procedures, and functions, distinct from every other activation.

6.2.3.3

The activation of a procedure or function shall be an activation of the block of the procedure-block of the procedure or function-block of the function, respectively, and shall be designated as *within*

a) the activation containing the procedure or function; and

b) all activations that that containing activation is within.

NOTE — An activation of a block B can only be within activations of blocks containing B. Thus, an activation is not within another activation of the same block.

Within an activation, an applied occurrence of a label or variable-identifier, or of a procedure-identifier or function-identifier local to the block of the activation, shall denote the corresponding program-point, variable, procedure, or function, respectively, of that activation; except that the function-identifier of an assignment-statement shall, within an activation of the function denoted by that function-identifier, denote the result of that activation.

6.2.3.4

A procedure-statement or function-designator contained in the algorithm of an activation and that specifies an activation of a block shall be designated the *activation-point* of the activation of the block.

6.2.3.5

All variables contained by an activation and any result of an activation, shall be totally-undefined at the commencement of that activation. The algorithm, program-points, variables, procedures, and functions, if any, shall exist until the termination of the activation.

6.3 Constant-definitions

A constant-definition shall introduce an identifier to denote a value.

```
constant-definition = identifier "=" constant .

constant = [ sign ] ( unsigned-number | constant-identifier )
           | character-string .

constant-identifier = identifier .
```

The occurrence of an identifier in a constant-definition of a constant-definition-part of a block shall constitute its defining-point for the region that is the block. The constant in a constant-definition shall not contain an applied occurrence of the identifier in the constant-definition. Each applied occurrence of that identifier shall be a constant-identifier and shall denote the value denoted by the constant of the constant-definition. A constant-identifier in a constant containing an occurrence of a sign shall have been defined to denote a value of real-type or of integer-type. The required constant-identifiers shall be as specified in **6.4.2.2** and **6.7.2.2**.

6.4 Type-definitions

6.4.1 General

A type-definition shall introduce an identifier to denote a type. Type shall be an attribute that is possessed by every value and every variable. Each occurrence of a new-type shall denote a type that is distinct from any other new-type.

type-definition = identifier "=" type-denoter .

type-denoter = type-identifier | new-type .

new-type = new-ordinal-type | new-structured-type | new-pointer-type .

The occurrence of an identifier in a type-definition of a type-definition-part of a block shall constitute its defining-point for the region that is the block. Each applied occurrence of that identifier shall be a type-identifier and shall denote the same type as that which is denoted by the type-denoter of the type-definition. Except for applied occurrences in the domain-type of a new-pointer-type, the type-denoter shall not contain an applied occurrence of the identifier in the type-definition.

Types shall be classified as simple-types, structured-types or pointer-types. The required type-identifiers and corresponding required types shall be as specified in **6.4.2.2** and **6.4.3.5**.

simple-type-identifier = type-identifier .

structured-type-identifier = type-identifier .

pointer-type-identifier = type-identifier .

type-identifier = identifier .

A type-identifier shall be considered as a simple-type-identifier, a structured-type-identifier, or a pointer-type-identifier, according to the type that it denotes.

6.4.2 Simple-types

6.4.2.1 General

A simple-type shall determine an ordered set of values. A value of an ordinal-type shall have an integer ordinal number; the ordering relationship between any two such values of one type shall be the same as that between their ordinal numbers. An ordinal-type-identifier shall denote an ordinal-type. A real-type-identifier shall denote the real-type.

simple-type = ordinal-type | real-type-identifier .

ordinal-type = new-ordinal-type | ordinal-type-identifier .

new-ordinal-type = enumerated-type | subrange-type .

ordinal-type-identifier = type-identifier .

real-type-identifier = type-identifier .

6.4.2.2 Required simple-types

The following types shall exist

a) *integer-type*. The required type-identifier **integer** shall denote the integer-type. The integer-type shall be an ordinal-type. The values shall be the whole numbers, denoted as specified in **6.1.5** by signed-integer (see also **6.7.2.2**). The ordinal number of a value of integer-type shall be the value itself.

b) *real-type*. The required type-identifier **real** shall denote the real-type. The real-type shall be a simple-type. The values shall be the real numbers, denoted as specified in **6.1.5** by signed-real.

c) *Boolean-type*. The required type-identifier **Boolean** shall denote the Boolean-type. The Boolean-type shall be an ordinal-type. The values shall be the enumeration of truth values denoted by the required constant-identifiers **false** and **true**, such that **false** is the predecessor of **true**. The ordinal numbers of the truth values denoted by **false** and **true** shall be the integer values 0 and 1 respectively.

d) *char-type*. The required type-identifier **char** shall denote the char-type. The char-type shall be an ordinal-type. The values shall be the enumeration of a set of implementation-defined characters, some possibly without graphic representations. The ordinal numbers of the character values shall be values of integer-type that are implementation-defined and that are determined by mapping the character values on to consecutive non-negative integer values starting at zero. The following relations shall hold.

   1) The subset of character values representing the digits 0 to 9 shall be numerically ordered and contiguous.

   2) The subset of character values representing the upper case letters A to Z, if available, shall be alphabetically ordered but not necessarily contiguous.

   3) The subset of character values representing the lower case letters a to z, if available, shall be alphabetically ordered but not necessarily contiguous.

NOTE — Operators applicable to the required simple-types are specified in **6.7.2**.

6.4.2.3 Enumerated-types

enumerated-type = "(" identifier-list ")" .

identifier-list = identifier { "," identifier } .

The occurrence of an identifier in the identifier-list of an enumerated-type shall constitute its defining-point for the region that is the block closest-containing the enumerated-type. Each applied occurrence of the identifier shall be a constant-identifier. Within an activation of the block, all applied occurrences of that identifier shall possess the type denoted by the enumerated-type and shall denote the type's value whose ordinal number is the number of occurrences of identifiers preceding that identifier in the identifier-list.

NOTE — Enumerated type constants are ordered by the sequence in which they are defined, and they have consecutive ordinal numbers starting at zero.

Examples:

```
(red, yellow, green, blue, tartan)
(club, diamond, heart, spade)
(married, divorced, widowed, single)
(scanning, found, notpresent)
(Busy, InterruptEnable, ParityError, OutOfPaper, LineBreak)
```

6.4.2.4 Subrange-types

A subrange-type shall include identification of the smallest and the largest value in the subrange. The first constant of a subrange-type shall specify the smallest value, and this shall be less than or equal to the largest value, which shall be specified by the second constant of the subrange-type. Both constants shall be of the same ordinal-type, and that ordinal-type shall be designated the *host-type* of the subrange-type.

subrange-type = constant ".." constant .

Examples:

```
1..100
-10..+10
red..green
'0'..'9'
```

6.4.3 Structured-types

6.4.3.1 General

A new-structured-type shall be classified as an array-type, record-type, or set-type according to the unpacked-structured-type closest-contained by the new-structured-type. A component of a value of a structured-type shall be a value.

structured-type = new-structured-type | structured-type-identifier .

new-structured-type = ["packed"] unpacked-structured-type .

unpacked-structured-type = array-type | record-type | set-type .

NOTE — The symbol **file-type** has been deleted.

The occurrence of the token packed in a new-structured-type shall designate the type denoted thereby as *packed*. The designation of a structured-type as packed shall indicate to the processor that data-storage of values should be economized, even if this causes operations on, or accesses to components of, variables possessing the type to be less efficient in terms of space or time.

The designation of a structured-type as packed shall affect the representation in data-storage of that structured-type only; i.e., if a component is itself structured, the component's representation in data-storage shall be packed only if the type of the component is designated packed.

NOTE — The ways in which the treatment of entities of a type is affected by whether or not the type is designated packed are specified in **6.4.3.2**, **6.4.5**, **6.6.3.3**, **6.6.3.8**, **6.6.5.4**, and **6.7.1**.

6.4.3.2 Array-types

An array-type shall be structured as a mapping from each value specified by its index-type to a distinct component. Each component shall have the type denoted by the type-denoter of the component-type of the array-type.

array-type = “array” “[” index-type { “,” index-type } “]” “of” component-type .

index-type = ordinal-type .

component-type = type-denoter .

Example 1:

```
array [1..100] of real
array [Boolean] of colour
```

An array-type that specifies a sequence of two or more index-types shall be an abbreviated notation for an array-type specified to have as its index-type the first index-type in the sequence and to have a component-type that is an array-type specifying the sequence of index-types without the first index-type in the sequence and specifying the same component-type as the original specification. The component-type thus constructed shall be designated *packed* if and only if the original array-type is designated packed. The abbreviated form and the full form shall be equivalent.

NOTE — 1 Each of the following two examples thus contains different ways of expressing its array-type.

Example 2:

```
array [Boolean] of array [1..10] of array [size] of real
array [Boolean] of array  [1..10, size] of real
array [Boolean, 1..10, size] of real
array [Boolean, 1..10] of array [size] of real
```

Example 3:

```
packed array [1..10, 1..8] of Boolean
packed array [1..10] of packed array [1..8] of Boolean
```

Let i denote a value of the index-type; let V_i denote a value of that component of the array-type that corresponds to the value i by the structure of the array-type; let the smallest and largest values specified by the index-type be denoted by m and n, respectively; and let k = (ord(n)-ord(m)+1) denote the number of values specified by the index-type; then the values of the array-type shall be the distinct k-tuples of the form

$(V_m,...,V_n)$.

NOTE — 2 A value of an array-type does not therefore exist unless all of its component-values are defined. If the component-type has c values, then it follows that the cardinality of the set of values of the array-type is c raised to the power k.

Any type designated packed and denoted by an array-type having as its index-type a denotation of a subrange-type specifying a smallest value of 1 and a largest value of greater than 1, and having as its component-type a denotation of the char-type, shall be designated a *string-type*.

The correspondence of character-strings to values of string-types is obtained by relating the individual string-elements of the character-string, taken in textual order, to the components of the values of the string-type in order of increasing index.

NOTE — 3 The values of a string-type possess additional properties which define their use with relational-operators (see **6.7.2.5**).

6.4.3.3 Record-types

The structure and values of a record-type shall be the structure and values of the field-list of the record-type.

```
record-type = "record" field-list "end" .

field-list = [ ( fixed-part [ ";" variant-part ] | variant-part ) [ ";" ] ] .

fixed-part = record-section { ";" record-section } .

record-section = identifier-list ":" type-denoter .

field-identifier = identifier .

variant-part = "case" variant-selector "of" variant { ";" variant } .

variant-selector = [ tag-field ":" ] tag-type .

tag-field = identifier .

variant = case-constant-list ":" "(" field-list ")" .

tag-type = ordinal-type-identifier .

case-constant-list = case-constant { "," case-constant } .

case-constant = constant .
```

A field-list containing neither a fixed-part nor a variant-part shall have no components, shall define a single null value, and shall be designated *empty*.

The occurrence of an identifier in the identifier-list of a record-section of a fixed-part of a field-list shall constitute its defining-point as a field-identifier for the region that is the record-type closest-containing the field-list and shall associate the field-identifier with a distinct component, which shall be designated a *field*, of the record-type and of the field-list. That component shall have the type denoted by the type-denoter of the record-section.

The field-list closest-containing a variant-part shall have a distinct component that shall have the values and structure defined by the variant-part.

Let V_i denote the value of the i-th component of a non-empty field-list having m components; then the values of the field-list shall be distinct m-tuples of the form

$$(V_1, V_2, \ldots, V_m).$$

NOTE — 1 If the type of the i-th component has F_i values, then the cardinality of the set of values of the field-list is $(F_1 * F_2 * \ldots * F_m)$.

A tag-type shall be the type denoted by the ordinal-type-identifier of the tag-type. A case-constant shall denote the value denoted by the constant of the case-constant.

The type of each case-constant in the case-constant-list of a variant of a variant-part shall be compatible with the tag-type of the variant-selector of the variant-part. The values denoted by all case-constants of a type that is required to be compatible with a given tag-type shall be distinct and the set thereof shall be equal to the set of values specified by the tag-type. The values denoted by the case-constants of the case-constant-list of a variant shall be designated as corresponding to the variant.

With each variant-part shall be associated a type designated the *selector-type* possessed by the variant-part. If the variant-selector of the variant-part contains a tag-field, or if the case-constant-list of each variant of the variant-part contains only one case-constant, then the selector-type shall be denoted by the tag-type, and each variant of the variant-part shall be associated with those values specified by the selector-type denoted by the case-constants of the case-constant-list of the variant. Otherwise, the selector-type possessed by the variant-part shall be a new ordinal-type that is constructed to possess exactly one value for each variant of the variant-part, and no others, and each such variant shall be associated with a distinct value of that type.

Each variant-part shall have a component which shall be designated the *selector* of the variant-part, and which shall possess the selector-type of the variant-part. If the variant-selector of the variant-part contains a tag-field, then the occurrence of an identifier in the tag-field shall constitute the defining-point of the identifier as a field-identifier for the region that is the record-type closest-containing the variant-part and shall associate the field-identifier with the selector of the variant-part. The selector shall be designated a *field* of the record-type if and only if it is associated with a field-identifier.

Each variant of a variant-part shall denote a distinct component of the variant-part; the component shall have the values and structure of the field-list of the variant, and shall be associated with those values specified by the selector-type possessed by the variant-part associated with the variant. The value of the selector of the variant-part shall cause the associated variant and component of the variant-part to be in a state that shall be designated *active*.

The values of a variant-part shall be the distinct pairs

$$(k, X_k)$$

where k represents a value of the selector of the variant-part, and X_k is a value of the field-list of the active variant of the variant-part.

NOTES

2 If there are n values specified by the selector-type, and if the field-list of the variant associated with the i-th value has T_i values, then the cardinality of the set of values of the variant-part is $(T_1 + T_2 + ... + T_n)$. There is no component of a value of a variant-part corresponding to any non-active variant of the variant-part.

3 Restrictions placed on the use of fields of a record-variable pertaining to variant-parts are specified in **6.5.3.3**, **6.6.3.3**, and **6.6.5.3**.

Examples:

```
      record
        year : 0..2000;
        month : 1..12;
        day : 1..31
      end

      record
        name, firstname : string;
        age : 0..99;
        case married : Boolean of
          true : (Spousesname : string);
          false : ( )
      end

      record
```

```
        x, y : real;
        area : real;
        case shape of
          triangle :
            (side : real; inclination, angle1, angle2 : angle);
          rectangle :
            (side1, side2 : real; skew : angle);
          circle :
            (diameter : real);
      end
```

6.4.3.4 Set-types

A set-type shall determine the set of values that is structured as the power set of the base-type of the set-type. Thus, each value of a set-type shall be a set whose members shall be unique values of the base-type.

set-type = "set" "of" base-type .

base-type = ordinal-type .

NOTE — 1 Operators applicable to values of set-types are specified in **6.7.2.4**.

Examples:

```
      set of char
      set of (club, diamond, heart, spade)
```

NOTE — 2 If the base-type of a set-type has b values, then the cardinality of the set of values is 2 raised to the power b.

For each ordinal-type T that is not a subrange-type, there shall exist both an unpacked set-type designated the *unpacked-canonical-set-of-T-type* and a packed set-type designated the *packed-canonical-set-of-T-type*. If S is any subrange-type and T is its host-type, then the set of values determined by the type set of S shall be included in the sets of values determined by the unpacked-canonical-set-of-T-type and by the packed-canonical-set-of-T-type (see **6.7.1**).

6.4.4 Pointer-types

The values of a pointer-type shall consist of a single *nil-value* and a set of *identifying-values* each identifying a distinct variable possessing the domain-type of the new-pointer-type. The set of identifying-values shall be dynamic, in that the variables and the values identifying them shall be permitted to be created and destroyed. Identifying-values and the variables identified by them shall be created only by the required procedure **new** (see **6.6.5.3**).

NOTE — 1 Since the nil-value is not an identifying-value, it does not identify a variable.

The token nil shall denote the nil-value in all pointer-types.

pointer-type = new-pointer-type | pointer-type-identifier .

new-pointer-type = "↑" domain-type .

domain-type = type-identifier .

NOTE — 2 The token nil does not have a single type, but assumes a suitable pointer-type to satisfy the assignment-compatibility rules, or the compatibility rules for operators, if possible.

6.4.5 Compatible types

Types T1 and T2 shall be designated *compatible* if any of the following four statements is true:

a) T1 and T2 are the same type.

b) T1 is a subrange of T2, or T2 is a subrange of T1, or both T1 and T2 are subranges of the same host-type.

c) T1 and T2 are set-types of compatible base-types, and either both T1 and T2 are designated packed or neither T1 nor T2 is designated packed.

d) T1 and T2 are string-types with the same number of components.

6.4.6 Assignment-compatibility

A value of type T2 shall be designated *assignment-compatible* with a type T1 if any of the following five statements is true:

a) T1 and T2 are the same type.

b) T1 is the real-type and T2 is the integer-type.

c) T1 and T2 are compatible ordinal-types, and the value of type T2 is in the closed interval specified by the type T1.

d) T1 and T2 are compatible set-types, and all the members of the value of type T2 are in the closed interval specified by the base-type of T1.

e) T1 and T2 are compatible string-types.

At any place where the rule of assignment-compatibility is used

a) it shall be an error if T1 and T2 are compatible ordinal-types and the value of type T2 is not in the closed interval specified by the type T1;

b) it shall be an error if T1 and T2 are compatible set-types and any member of the value of type T2 is not in the closed interval specified by the base-type of the type T1.

At any place where the rule of assignment-compatibility is used to require a value of integer-type to be assignment-compatible with a real-type, an implicit integer-to-real conversion shall be performed.

6.4.7 Example of a type-definition-part

```
type
 count = integer;
 range = integer;
 colour = (red, yellow, green, blue);
 sex = (male, female);
 year = 1900..1999;
 shape = (triangle, rectangle, circle);
```

```
punchedcard = array [1..80] of char;
polar = record
             r : real;
         theta : angle
        end;
indextype = 1..limit;
vector = array [indextype] of real;

person = ↑ persondetails;

persondetails = record
                 name, firstname : charsequence;
                 age : natural;
                 married : Boolean;
                 father, child, sibling : person;
                 case s : sex of
                  male :
                   (enlisted, bearded : Boolean);
                  female :
                   (mother, programmer : Boolean)
                end;
```

NOTE — In the above example *count*, *range*, and **integer** denote the same type. The types denoted by *year* and *natural* are compatible with, but not the same as, the type denoted by *range*, *count*, and **integer**.

6.5 Declarations and denotations of variables

6.5.1 Variable-declarations

A variable shall be an entity to which a value can be attributed (see **6.8.2.2**). Each identifier in the identifier-list of a variable-declaration shall denote a distinct variable possessing the type denoted by the type-denoter of the variable-declaration.

variable-declaration = identifier-list ":" type-denoter .

The occurrence of an identifier in the identifier-list of a variable-declaration of the variable-declaration-part of a block shall constitute its defining-point as a variable-identifier for the region that is the block. The structure of a variable possessing a structured-type shall be the structure of the structured-type. A use of a variable-access shall be an access, at the time of the use, to the variable thereby denoted. A variable-access, according to whether it is an entire-variable, a component-variable, an identified-variable, or a buffer-variable, shall denote a declared variable, a component of a variable, a variable that is identified by a pointer value (see **6.4.4**), or a buffer-variable, respectively.

variable-access = entire-variable | component-variable | identified-variable .

NOTE — The symbol **buffer-variable** has been deleted.

Example of a variable-declaration-part:

```
var
```

```
x, y, z, max : real;
i, j : integer;
k : 0..9;
p, q, r : Boolean;
operator : (plus, minus, times);
a : array [0..63] of real;
c : colour;
hue1, hue2 : set of colour;
p1, p2 : person;
m, m1, m2 : array [1..10, 1..10] of real;
coord : polar;

date : record
         month : 1..12;
         year : integer
       end;
```

NOTE — Variables occurring in examples in the remainder of this International Standard should be assumed to have been declared as specified in the above example.

6.5.2 Entire-variables

entire-variable = variable-identifier .

variable-identifier = identifier .

6.5.3 Component-variables

6.5.3.1 General

A component of a variable shall be a variable. A component-variable shall denote a component of a variable. A reference or an access to a component of a variable shall constitute a reference or an access, respectively, to the variable. The value, if any, of the component of a variable shall be the same component of the value, if any, of the variable.

component-variable = indexed-variable | field-designator .

6.5.3.2 Indexed-variables

A component of a variable possessing an array-type shall be denoted by an indexed-variable.

indexed-variable = array-variable "[" index-expression, { "," index-expression } "]" .

array-variable = variable-access .

index-expression = expression .

An array-variable shall be a variable-access that denotes a variable possessing an array-type. For an indexed-variable closest-containing a single index-expression, the value of the index-expression shall be assignment-compatible with the index-type of the array-type. The component denoted by the indexed-variable shall be the component that corresponds to the value of the index-expression by the mapping of the type possessed by the array-variable (see **6.4.3.2**).

Example 1:

```
        a[12]
        a[i + j]
        m[k]
```

If the array-variable is itself an indexed-variable, an abbreviation shall be permitted. In the abbreviated form, a single comma shall replace the sequence *] [* that occurs in the full form. The abbreviated form and the full form shall be equivalent.

The order of both the evaluation of the index-expressions of, and the access to the array-variable of, an indexed-variable shall be implementation-dependent.

Example 2:

```
        m[k][1]
        m[k, 1]
```

NOTE — These two examples denote the same component-variable.

6.5.3.3 Field-designators

A field-designator either shall denote that component of the record-variable of the field-designator associated (see **6.4.3.3**) with the field-identifier of the field-specifier of the field-designator or shall denote the variable denoted by the field-designator-identifier (see **6.8.3.10**) of the field-designator. A record-variable shall be a variable-access that denotes a variable possessing a record-type.

The occurrence of a record-variable in a field-designator shall constitute the defining-point of the field-identifiers associated with components of the record-type possessed by the record-variable, for the region that is the field-specifier of the field-designator.

field-designator = record-variable "." field-specifier | field-designator-identifier .

record-variable = variable-access .

field-specifier = field-identifier .

Examples:

```
        p2↑.mother
        coord.theta
```

An access to a component of a variant of a variant-part, where the selector of the variant-part is not a field, shall attribute to the selector that value associated (see **6.4.3.3**) with the variant. It shall be an error unless a variant is active for the entirety of each reference and access to each component of the variant.

When a variant becomes non-active, all of its components shall become totally-undefined.

NOTE — If the selector of a variant-part is undefined, then no variant of the variant-part is active.

6.5.4 Identified-variables

An identified-variable shall denote the variable, if any, identified by the value of the pointer-variable of the identified-variable (see **6.4.4** and **6.6.5.3**) shall be accessible until the variable is made inaccessible (see the required procedure **dispose**, **6.6.5.3**).

NOTE — The accessibility of the variable also depends on the existence of a pointer-variable that has attributed to it the corresponding identifying-value.

A pointer-variable shall be a variable-access that denotes a variable possessing a pointer-type. It shall be an error if the pointer-variable of an identified-variable either denotes a nil-value or is undefined. It shall be an error to remove from the set of values of the pointer-type the identifying-value of an identified-variable (see **6.6.5.3**) when a reference to the identified-variable exists.

Examples:

```
p1↑
p1↑.father↑
p1↑.sibling↑.father↑
```

6.6 Procedure and function declarations

6.6.1 Procedure-declarations

```
procedure-declaration =  procedure-heading ";" directive
                        | procedure-identification ";" procedure-block
                        | procedure-heading ";" procedure-block .

procedure-heading = "procedure" identifier [ formal-parameter-list ] .

procedure-identification = "procedure" procedure-identifier .

procedure-identifier = identifier .

procedure-block = block .
```

The occurrence of a formal-parameter-list in a procedure-heading of a procedure-declaration shall define the formal-parameters of the procedure-block, if any, associated with the identifier of the procedure-heading to be those of the formal-parameter-list.

The occurrence of an identifier in the procedure-heading of a procedure-declaration shall constitute its defining-point as a procedure-identifier for the region that is the block closest-containing the procedure-declaration.

Each identifier having a defining-point as a procedure-identifier in a procedure-heading of a procedure-declaration in which the directive **forward** occurs shall have exactly one of its applied occurrences in a procedure-identification of a procedure-declaration, and this applied occurrence shall be closest-contained by the procedure-and-function-declaration-part closest-containing the procedure-heading.

The occurrence of a procedure-block in a procedure-declaration shall associate the procedure-block with the identifier in the procedure-heading, or with the procedure-identifier in the procedure-identification, of the procedure-declaration.

There shall be at most one procedure-block associated with a procedure-identifier.

Example of procedure-and-function-declaration-part:

```
procedure bisect (function f(x : real) : real;
                            a, b       : real;
                  var       result     : real);
 {This procedure attempts to find a zero of f(x) in (a,b) by
  the method of bisection. It is assumed that the procedure is
  called with suitable values of a and b such that
      (f(a) < 0) and (f(b) >= 0)
  The estimate is returned in the last parameter.}
 const
   eps = 1e-10;
 var
   midpoint : real;
 begin
  {The invariant P is true by calling assumption}
  midpoint := a;
  while abs(a - b) > eps * abs(a) do begin
   midpoint := (a + b) / 2;
   if f(midpoint) < 0 then a := midpoint
   else b := midpoint
  {Which re-establishes the invariant:
     P = (f(a) < 0) and (f(b) >= 0)
   and reduces the interval (a,b) provided that the
   value of midpoint is distinct from both a and b.}
 end;
  {P together with the loop exit condition assures that a zero
   is contained in a small subinterval.  Return the midpoint as
   the zero.}
  result := midpoint
end;
```

6.6.2 Function-declarations

function-declaration = function-heading ";" directive
| function-identification ";" function-block
| function-heading ";" function-block .

function-heading = "function" identifier [formal-parameter-list] ":" result-type .

function-identification = "function" function-identifier .

function-identifier = identifier .

result-type = simple-type-identifier | pointer-type-identifier .

function-block = block .

NOTE — See clause 8 for **result-type** production rule.

The occurrence of a formal-parameter-list in a function-heading of a function-declaration shall define the formal-parameters of the function-block, if any, associated with the identifier of the function-heading to be those of the formal-parameter-list. The function-block shall contain at least one assignment-statement such that the function-identifier of the assignment-statement is associated with the block (see **6.8.2.2**).

The occurrence of an identifier in the function-heading of a function-declaration shall constitute its defining-point as a function-identifier associated with the result type denoted by the result-type for the region that is the block closest-containing the function-declaration.

Each identifier having a defining-point as a function-identifier in the function-heading of a function-declaration in which the directive **forward** occurs shall have exactly one of its applied occurrences in a function-identification of a function-declaration, and this applied occurrence shall be closest-contained by the procedure-and-function-declaration-part closest-containing the function-heading.

The occurrence of a function-block in a function-declaration shall associate the function-block with the identifier in the function-heading, or with the function-identifier in the function-identification, of the function-declaration; the block of the function-block shall be associated with the result type that is associated with the identifier or function-identifier.

There shall be at most one function-block associated with a function-identifier.

Example of a procedure-and-function-declaration-part:

```
function Sqrt (x : real) : real;
{This function computes the square root of x (x > 0) using Newton's
 method.}
var
   old, estimate : real;
begin
 estimate := x;
 repeat
   old := estimate;
   estimate := (old + x / old) * 0.5;
 until abs(estimate - old) < eps * estimate;
 {eps being a global constant}
 Sqrt := estimate
 end { of Sqrt };

 function max (a : vector) : real;
 {This function finds the largest component of the value of a.}
 var
    largestsofar : real;
    fence : indextype;
 begin
  largestsofar := a[1];
  {Establishes largestsofar = max(a[1])}
  for fence := 2 to limit do begin
    if largestsofar < a[fence] then largestsofar := a[fence]
    {Re-establishing largestsofar = max(a[1], ... ,a[fence])}
   end;
   {So now largestsofar = max(a[1], ... ,a[limit])}
   max := largestsofar
  end { of max };

  function GCD (m, n : natural) : natural;
  begin
    if n=0 then GCD := m else GCD := GCD(n, m mod n);
  end;


 {The following two functions analyze a parenthesized expression and
  convert it to an internal form. They are declared forward
  since they are mutually recursive, i.e., they call each other.
  These function-declarations use the following identifiers that are not
  defined in this International Standard: formula, IsOpenParenthesis, IsOperator,
  MakeFormula, nextsym, operation, ReadElement, ReadOperator, and
  SkipSymbol. }

 function ReadExpression : formula; forward;

 function ReadOperand : formula; forward;

 function ReadExpression; {See forward declaration of heading.}
 var
   this : formula;
   op : operation;
 begin
  this := ReadOperand;
  while IsOperator(nextsym) do
   begin
    op := ReadOperator;
    this := MakeFormula(this, op, ReadOperand);
   end;
  ReadExpression := this
 end;


 function ReadOperand; {See forward declaration of heading.}
 begin
  if IsOpenParenthesis(nextsym) then
   begin
   SkipSymbol;
   ReadOperand := ReadExpression;
   {nextsym should be a close-parenthesis}
   SkipSymbol
   end
  else ReadOperand := ReadElement
  end;
```

6.6.3 Parameters

6.6.3.1 General

The identifier-list in a value-parameter-specification shall be a list of value parameters. The identifier-list in a variable-parameter-specification shall be a list of variable parameters.

```
formal-parameter-list = "(" formal-parameter-section { ";" formal-parameter-section } ")" .

formal-parameter-section = value-parameter-specification
                         | variable-parameter-specification
                         | procedural-parameter-specification
                         | functional-parameter-specification .

value-parameter-specification = identifier-list ":" type-identifier .

variable-parameter-specification = "var" identifier-list ":" type-identifier .

procedural-parameter-specification = procedure-heading .

functional-parameter-specification = function-heading .
```

An identifier defined to be a parameter-identifier for the region that is the formal-parameter-list of a procedure-heading shall be designated a *formal-parameter* of the block of the procedure-block, if any, associated with the identifier of the procedure-heading. An identifier defined to be a parameter-identifier for the region that is the formal-parameter-list of a function-heading shall be designated a *formal-parameter* of the block of the function-block, if any, associated with the identifier of the function-heading.

The occurrence of an identifier in the identifier-list of a value-parameter-specification or a variable-parameter-specification shall constitute its defining-point as a parameter-identifier for the region that is the formal-parameter-list closest-containing it, and its defining-point as the associated variable-identifier for the region that is the block, if any, of which it is a formal-parameter.

The occurrence of the identifier of a procedure-heading in a procedural-parameter-specification shall constitute its defining-point as a parameter-identifier for the region that is the formal-parameter-list closest-containing it, and its defining-point as the associated procedure-identifier for the region that is the block, if any, of which it is a formal-parameter.

The occurrence of the identifier of a function-heading in a functional-parameter-specification shall constitute its defining-point as a parameter-identifier for the region that is the formal-parameter-list closest-containing it, and its defining-point as the associated function-identifier for the region that is the block, if any, of which it is a formal-parameter.

NOTE — 2 If the formal-parameter-list is contained in a procedural-parameter-specification or a functional-parameter-specification, there is no corresponding procedure-block or function-block.

6.6.3.2 Value parameters

The formal-parameter and its associated variable-identifier shall denote the same variable. The formal-parameter shall possess the type denoted by the type-identifier of the value-parameter-specification. The type possessed by the formal-parameter shall be one that is permitted as the component-type of a file-type (see **6.4.3.5**). The actual-parameter (see **6.7.3** and **6.8.2.3**) shall be an expression whose value is assignment-compatible with the type possessed by the formal-parameter. The current value of the expression shall be attributed upon activation of the block to the variable that is denoted by the formal-parameter.

6.6.3.3 Variable parameters

The actual-parameter shall be a variable-access. The type possessed by the actual-parameters shall be the same as that denoted by the type-identifier of the variable-parameter-specification, and the formal-parameters shall also possess that type. The actual-parameter shall be accessed before the activation of the block, and this access shall establish a reference to the variable thereby accessed during the entire activation of the block; the corresponding formal-parameter and its associated variable-identifier shall denote the referenced variable during the activation.

An actual variable parameter shall not denote a field that is the selector of a variant-part. An actual variable parameter shall not denote a component of a variable where that variable possesses a type that is designated packed.

6.6.3.4 Procedural parameters

The actual-parameter (see **6.7.3** and **6.8.2.3**) shall be a procedure-identifier that has a defining-point contained by its block. The formal-parameter-list, if any, closest-contained by the formal-parameter-section and the formal-parameter-list, if any, that defines the formal-parameters of the procedure denoted by the actual-parameter shall be congruous, or neither formal-parameter-list shall occur. The formal-parameter and its associated procedure-identifier shall denote the actual-parameter during the entire activation of the block.

6.6.3.5 Functional parameters

The actual-parameter (see **6.7.3** and **6.8.2.3**) shall be a function-identifier that has a defining-point contained by its block. The formal-parameter-list, if any, closest-contained by the formal-parameter-section and the formal-parameter-list, if any, that defines the formal-parameters of the function denoted by the actual-parameter shall be congruous, or neither formal-parameter-list shall occur. The result-type closest-contained by the formal-parameter-section shall denote the same type as the result type of the function. The formal-parameter and its associated function-identifier shall denote the actual-parameter during the entire activation of the block.

6.6.3.6 Parameter list congruity

Two formal-parameter-lists shall be congruous if they contain the same number of formal-parameter-sections and if the formal-parameter-sections in corresponding positions match. Two formal-parameter-sections shall match if any of the following statements is true.

a) They are both value-parameter-specifications containing the same number of parameters and the type-identifier in each value-parameter-specification denotes the same type.

b) They are both variable-parameter-specifications containing the same number of parameters and the type-identifier in each variable-parameter-specification denotes the same type.

c) They are both procedural-parameter-specifications and the formal-parameter-lists of the procedure-headings thereof are congruous.

d) They are both functional-parameter-specifications, the formal-parameter-lists of the function-headings thereof are congruous, and the type-identifiers of the result-types of the function-headings thereof denote the same type.

6.6.4 Required procedures and functions

The required procedure-identifiers and function-identifiers and the corresponding required procedures and functions shall be as specified in **6.6.5**, **6.6.6**, and **6.9**.

NOTE — Required procedures and functions do not necessarily follow the rules given elsewhere for procedures and functions.

6.6.5 Required procedures

6.6.5.1 General

The required procedures shall be dynamic allocation procedures and transfer procedures.

6.6.5.3 Dynamic allocation procedures

new(p)
: shall create a new variable that is totally-undefined, shall create a new identifying-value of the pointer-type associated with p, that identifies the new variable, and shall attribute this identifying-value to the variable denoted by the variable-access p. The created variable shall possess the type that is the domain-type of the pointer-type possessed by p.

new(p,c_1,...,c_n)
: shall create a new variable that is totally-undefined, shall create a new identifying-value of the pointer-type associated with p, that identifies the new variable, and shall attribute this identifying-value to the variable denoted by the variable-access p. The created variable shall possess the record-type that is the domain-type of the pointer-type possessed by p and shall have nested variants that correspond to the case-constants c_1,...,c_n. The case-constants shall be listed in order of increasing nesting of the variant-parts. Any variant not specified shall be at a deeper level of nesting than that specified by cn.

  It shall be an error if a variant of a variant-part within the new variable is active and a different variant of the variant-part is one of the specified variants.

dispose(q)
: shall remove the identifying-value denoted by the expression q from the pointer-type of q. It shall be an error if the identifying-value had been created using the form new(p,c_1,...,c_n).

dispose(q,k_1,...,k_m)
: shall remove the identifying-value denoted by the expression q from the pointer-type of q. The case-constants k_1,...,k_m shall be listed in order of increasing nesting of the variant-parts. It shall be an error unless the variable had been created using the form new(p,c_1,...,c_n) and m is equal to n. It shall be an error if the variants in the variable identified by the pointer value of q are different from those specified by the values denoted by the case-constants k_1,...,k_m.

NOTE — The removal of an identifying-value from the pointer-type to which it belongs renders the identified-variable inaccessible (see **6.5.4**) and makes undefined all variables and functions that have that value attributed (see **6.6.3.2** and **6.8.2.2**).

It shall be an error if q has a nil-value or is undefined.

It shall be an error if a variable created using the second form of new is accessed by the identified-variable of the variable-access of a factor, of an assignment-statement, or of an actual-parameter.

6.6.5.4 Transfer procedures

In the statement pack(a,i,z) and in the statement unpack(z,a,i) the following shall hold: a and z shall be variable-accesses; a shall possess an array-type not designated packed; z shall possess an array-type designated packed; the component-types of the types of a and z shall be the same; and the value of the expression i shall be assignment-compatible with the index-type of the type of a.

Let j and k denote auxiliary variables that the program does not otherwise contain and that have the type that is the index-type of the type of z and a, respectively. Let u and v denote the smallest and largest values of the index-type of the type of z. Each of the statements pack(a,i,z) and unpack(z,a,i) shall establish references to the variables denoted by a and z for the remaining execution of the statements; let aa and zz, respectively, denote the referenced variables within the following sentence. Then the statement pack(a,i,z) shall be equivalent to

```
begin
k := i;
for j := u to v do
  begin
  zz[j] := aa[k];
  if j <> v then k := succ(k)
  end
end
```

and the statement unpack(z,a,i) shall be equivalent to

```
begin
k := i;
for j := u to v do
  begin
  aa[k] := zz[j];
  if j <> v then k := succ(k)
  end
end
```

NOTE — Errors will arise if the references cannot be established, if one or more of the values attributed to j is not assignment-compatible with the index-type of the type of a, or if an evaluated array component is undefined.

6.6.6 Required functions

6.6.6.1 General

The required functions shall be arithmetic functions, transfer functions, ordinal functions, and Boolean functions.

6.6.6.2 Arithmetic functions

For the following arithmetic functions, the expression x shall be either of real-type or integer-type. For the functions **abs** and **sqr**, the type of the result shall be the same as the type of the parameter, x. For the remaining arithmetic functions, the result shall always be of real-type. The result shall be as shown in table 2.

Table 1 — Arithmetic function results

Function	Result
abs(x)	absolute value of x
sqr(x)	square of x It shall be an error if such a value does not exist.
sin(x)	sine of x, where x is in radians
cos(x)	cosine of x, where x is in radians
exp(x)	base of natural logarithms raised to the power x
ln(x)	natural logarithm of x, if x is greater than zero It shall be an error if x is not greater than zero.
sqrt(x)	non-negative square root of x, if x is not negative It shall be an error if x is negative.
arctan(x)	principal value, in radians, of the arctangent of x

6.6.6.3 Transfer functions

trunc(x)

From the expression x that shall be of real-type, this function shall return a result of integer-type. The value of trunc(x) shall be such that if x is positive or zero, then $0 \le x - \text{trunc}(x) < 1$; otherwise, $-1 < x - \text{trunc}(x) \le 0$. It shall be an error if such a value does not exist.

Examples:

```
trunc(3.5)  {yields 3}
trunc(-3.5) {yields -3}
```

round(x)

From the expression x that shall be of real-type, this function shall return a result of integer-type. If x is positive or zero, round(x) shall be equivalent to trunc(x+0.5); otherwise, round(x) shall be equivalent to trunc(x−0.5). It shall be an error if such a value does not exist.

Examples:

```
round(3.5)    {yields 4}
round(-3.5)   {yields -4}
```

6.6.6.4 Ordinal functions

ord(x)

From the expression x that shall be of an ordinal-type, this function shall return a result of integer-type that shall be the ordinal number (see **6.4.2.2** and **6.4.2.3**) of the value of the expression x.

chr(x)

From the expression x that shall be of integer-type, this function shall return a result of char-type that shall be the value whose ordinal number is equal to the value of the expression x, if such a character value exists. It shall be an error if such a character value does not exist. For any value, ch, of char-type, it shall be true that

chr(ord(ch)) = ch

succ(x)

From the expression x that shall be of an ordinal-type, this function shall return a result that shall be of the same type as that of the expression (see **6.7.1**). The function shall yield a value whose ordinal number is one greater than that of the expression x, if such a value exists. It shall be an error if such a value does not exist.

pred(x)
From the expression x that shall be of an ordinal-type, this function shall return a result that shall be of the same type as that of the expression (see **6.7.1**). The function shall yield a value whose ordinal number is one less than that of the expression x, if such a value exists. It shall be an error if such a value does not exist.

6.6.6.5 Boolean functions

odd(x)
From the expression x that shall be of integer-type, this function shall be equivalent to the expression

```
(abs(x) mod 2 = 1).
```

6.7 Expressions

6.7.1 General

An expression shall denote a value. The use of a variable-access as a factor shall denote the value, if any, attributed to the variable accessed thereby. When a factor is used, it shall be an error if the variable denoted by a variable-access of the factor is undefined. Operator precedences shall be according to four classes of operators as follows. The operator not shall have the highest precedence, followed by the multiplying-operators, then the adding-operators and signs, and finally, with the lowest precedence, the relational-operators. Sequences of two or more operators of the same precedence shall be left associative.

```
expression = simple-expression [ relational-operator simple-expression ] .

simple-expression = [ sign ] term { adding-operator term } .

term = factor { multiplying-operator factor } .

factor = variable-access | unsigned-constant | function-designator
       | set-constructor | "(" expression ")" | "not" factor .

unsigned-constant = unsigned-number | character-string | constant-identifier | "nil" .

set-constructor = "[" [ member-designator { "," member-designator } ] "]" .

member-designator = expression [ ".." expression ] .
```

Any factor whose type is S, where S is a subrange of T, shall be treated as if it were of type T. Similarly, any factor whose type is set of S shall be treated as if it were of the unpacked-canonical-set-of-T-type, and any factor whose type is packed set of S shall be treated as of the packed-canonical-set-of-T-type.

A set-constructor shall denote a value of a set-type. The set-constructor *[ ]* shall denote the value in every set-type that contains no members. A set-constructor containing one or more member-designators shall denote either a value of the unpacked-canonical-set-of-T-type or, if the context so requires, the packed-canonical-set-of-T-type, where T is the type of every expression of each member-designator of the set-constructor. The type T shall be an ordinal-type. The value denoted by the set-constructor shall contain zero or more members, each of which shall be denoted by at least one member-designator of the set-constructor.

The member-designator x, where x is an expression, shall denote the member that shall be the value of x. The member-designator x..y, where x and y are expressions, shall denote zero or more members that shall be the values of the base-type in the closed interval from the value of x to the value of y. The order of evaluation of the expressions of a member-designator shall be implementation-dependent. The order of evaluation of the member-designators of a set-constructor shall be implementation-dependent.

NOTES

2 The member-designator x..y denotes no members if the value of x is greater than the value of y.

3 The set-constructor [ ] does not have a single type, but assumes a suitable type to satisfy the assignment-compatibility rules, or the compatibility rules for operators, if possible.

Examples:

```
 a) Factors:
                                15
                                (x + y + z)
                                sin(x + y)
                                [red, c, green]
                                [1, 5, 10..19, 23]
                                not p
 b) Terms:
                                x * y
                                i / (1 - i)
                                (x <= y) and (y < z)

 c) Simple Expressions:
                                p or q
                                x + y
                                -x
                                hue1 + hue2
                                i * j + 1

 d) Expressions:
                                x = 1.5
                                p <= q
                                p = q and r
                                (i < j) = (j < k)
                                c in hue1
```

6.7.2 Operators

6.7.2.1 General

multiplying-operator = "*" | "/" | "div" | "mod" | "and" .

adding-operator = "+" | "–" | "or" .

relational-operator = "=" | "<>" | "<" | ">" | "<=" | ">=" | "in" .

A factor, a term, or a simple-expression shall be designated an operand. The order of evaluation of the operands of

Table 2 — Dyadic arithmetic operations

Operator	Operation	Type of operands	Type of result
+	Addition	integer-type or real-type	integer-type if both operands are of integer-type otherwise real-type
–	Subtraction	integer-type or real-type	
*	Multiplication	integer-type or real-type	
/	Division	integer-type or real-type	real-type
div	Division with truncation	integer-type	integer-type
mod	Modulo	integer-type	integer-type

Table 3 — Monadic arithmetic operations

Operator	Operation	Type of operand	Type of result
+	Identity	integer-type real-type	integer-type real-type
–	Sign-inversion	integer-type real-type	integer-type real-type

a dyadic operator shall be implementation-dependent.

NOTE — This means, for example, that the operands may be evaluated in textual order, or in reverse order, or in parallel, or they may not both be evaluated.

6.7.2.2 Arithmetic operators

The types of operands and results for dyadic and monadic operations shall be as shown in tables 3 and 4 respectively.

NOTE — 1 The symbols +, –, and * are also used as set operators (see **6.7.2.4**).

A term of the form x/y shall be an error if y is zero; otherwise, the value of x/y shall be the result of dividing x by y.

A term of the form i div j shall be an error if j is zero; otherwise, the value of i div j shall be such that

```
abs(i) – abs(j) < abs((i div j) * j) <= abs(i)
```

where the value shall be zero if abs(i) < abs(j); otherwise, the sign of the value shall be positive if i and j have the same sign and negative if i and j have different signs.

A term of the form i mod j shall be an error if j is zero or negative; otherwise, the value of i mod j shall be that value of (i–(k*j)) for integral k such that 0 <= i mod j < j.

NOTE — 2 Only for i >= 0 and j > 0 does the relation (i div j) * j + i mod j = i hold.

The results of the real arithmetic operators and functions shall be the corresponding mathematical results.

Table 4 — Set operations

Operator	Operation	Type of operands	Type of result
+	Set union	The same unpacked-canonical-set-of-T-type or packed-canonical-set-of-T-type (see **6.7.1**)	Same as operands
–	Set difference		
*	Set intersection		

Table 5 — Relational operations

Operator	Type of operands	Type of result
= <>	Any simple-type, pointer-type, string-type, unpacked-canonical-set-of-T-type or packed-canonical-set-of-T-type	Boolean-type
< >	Any simple-type or string-type	Boolean-type
<= >=	Any simple-type, string-type, unpacked-canonical-set-of-T-type or packed-canonical-set-of-T-type	Boolean-type
in	Left operand: any ordinal-type T right operand: the unpacked-canonical-set-of-T-type or packed-canonical-set-of-T-type	Boolean-type

It shall be an error if an integer operation or function is not performed according to the mathematical rules for integer arithmetic.

6.7.2.3 Boolean operators

Operands and results for Boolean operations shall be of Boolean-type. The Boolean operators or, and, and not shall denote respectively the logical operations of disjunction, conjunction, and negation.

Boolean-expression = expression .

A Boolean-expression shall be an expression that denotes a value of Boolean-type.

6.7.2.4 Set operators

The types of operands and results for set operations shall be as shown in table 5.

Where x denotes a value of the ordinal-type T and u and v are operands of a canonical-set-of-T-type, it shall be true for all x that

— x is a member of the value u+v if and only if it is a member of the value of u or a member of the value of v;

— x is a member of the value u−v if and only if it is a member of the value of u and not a member of the value of v;

— x is a member of the value u*v if and only if it is a member of the value of u and a member of the value of v.

6.7.2.5 Relational operators

The types of operands and results for relational operations shall be as shown in table 6.

The operands of =, <>, <, >, >=, and <= shall be of compatible types, or they shall be of the same unpacked-canonical-set-of-T-type or packed-canonical-set-of-T-type, or one operand shall be of real-type and the other shall be of integer-type.

The operators =, <>, <, and > shall stand for *equal to*, *not equal to*, *less than*, and *greater than* respectively.

Except when applied to sets, the operators <= and >= shall stand for *less than or equal to* and *greater than or equal to* respectively. Where u and v denote operands of a set-type, u <= v shall denote the inclusion of u in v and u >= v shall denote the inclusion of v in u.

NOTE — Since the Boolean-type is an ordinal-type with false less than true, then if p and q are operands of Boolean-type, p = q denotes their equivalence and p <= q means p implies q.

When the relational-operators = , <> , < , > , <=, and >= are used to compare operands of compatible string-types (see **6.4.3.2**), they shall denote the lexicographic relations defined below. This lexicographic ordering shall impose a total ordering on values of a string-type.

If s1 and s2 are two values of compatible string-types and n denotes the number of components of the compatible string-types, then

```
s1 = s2  iff for all i in [1..n]: s1[i] = s2[i]

s1 < s2  iff there exists a p in [1..n]:
           (for all i in [1..p-1]:
             s1[i] = s2[i]) and s1[p] < s2[p]
```

The definitions of operations >, <>, <=, and >= are derived from the definitions of = and <.

The operator `in` shall yield the value true if the value of the operand of ordinal-type is a member of the value of the set-type; otherwise, it shall yield the value false.

6.7.3 Function-designators

A function-designator shall specify the activation of the block of the function-block associated with the function-identifier of the function-designator and shall yield the value of the result of the activation upon completion of the algorithm of the activation; it shall be an error if the result is undefined upon completion of the algorithm.

NOTE — When a function activation is terminated by a goto-statement (see **6.8.2.4**), the algorithm of the activation does not complete (see **6.2.3.2 a)**), and thus there is no error if the result of the activation is undefined.

If the function has any formal-parameters, there shall be an actual parameter-list in the function-designator. The actual-parameter-list shall be the list of actual-parameters that shall be bound to their corresponding formal-parameters defined in the function-declaration. The correspondence shall be established by the positions of the parameters in the lists of actual-parameters and formal-parameters respectively. The number of actual-parameters shall be equal to the number of formal-parameters. The types of the actual-parameters shall correspond to the types of the formal-parameters as specified by **6.6.3**. The order of evaluation, accessing, and binding of the actual-parameters shall be implementation-dependent.

```
function-designator = function-identifier [ actual-parameter-list ] .

actual-parameter-list = "(" actual-parameter { "," actual-parameter } ")" .
```

actual-parameter = expression | variable-access | procedure-identifier
| function-identifier .

Examples:

```
Sum(a, 63)
GCD(147, k)
sin(x + y)
ord(f↑)
```

6.8 Statements

6.8.1 General

Statements shall denote algorithmic actions and shall be executable.

NOTE — 1 A statement may be prefixed by a label.

A label, if any, of a statement S shall be designated as *prefixing* S. The label shall be permitted to occur in a goto-statement G (see **6.8.2.4**) if and only if any of the following three conditions is satisfied.

a) S contains G.

b) S is a statement of a statement-sequence containing G.

c) S is a statement of the statement-sequence of the compound-statement of the statement-part of a block containing G.

statement = [label ":"] (simple-statement | structured-statement) .

NOTE — 2 A goto-statement within a block may refer to a label in an enclosing block, provided that the label prefixes a statement at the outermost level of nesting of the block.

6.8.2 Simple-statements

6.8.2.1 General

A simple-statement shall be a statement not containing a statement. An empty-statement shall contain no symbol and shall denote no action.

simple-statement = empty-statement | assignment-statement
| procedure-statement | goto-statement .

empty-statement = .

6.8.2.2 Assignment-statements

An assignment-statement shall attribute the value of the expression of the assignment-statement either to the variable denoted by the variable-access of the assignment-statement or to the activation result that is denoted by the function-identifier of the assignment-statement; the value shall be assignment-compatible with the type possessed, respectively, by the variable or by the activation result. The function-block associated (see **6.6.2**) with the function-identifier of an assignment-statement shall contain the assignment-statement.

assignment-statement = (variable-access | function-identifier) ":=" expression .

The variable-access shall establish a reference to the variable during the execution of the assignment-statement. The order of establishing the reference to the variable and evaluating the expression shall be implementation-dependent.

The state of a variable or activation result when the variable or activation result does not have attributed to it a value specified by its type shall be designated *undefined.* If a variable possesses a structured-type, the state of the variable when every component of the variable is totally-undefined shall be designated *totally-undefined.* Totally-undefined shall be synonymous with undefined for an activation result or a variable that does not possess a structured-type.

Examples:

```
x := y + z
p := (1 <= i) and (i < 100)
i := sqr(k) - (i * j)
hue1 := [blue, succ(c)]
p1↑.mother := true
```

6.8.2.3 Procedure-statements

A procedure-statement shall specify the activation of the block of the procedure-block associated with the procedure-identifier of the procedure-statement. If the procedure has any formal-parameters, the procedure-statement shall contain an actual-parameter-list, which is the list of actual-parameters that shall be bound to their corresponding formal-parameters defined in the procedure-declaration. The correspondence shall be established by the positions of the parameters in the lists of actual-parameters and formal-parameters respectively. The number of actual-parameters shall be equal to the number of formal-parameters. The types of the actual-parameters shall correspond to the types of the formal-parameters as specified by **6.6.3**.

The order of evaluation, accessing, and binding of the actual-parameters shall be indeterminate.

procedure-statement = procedure-identifier ([actual-parameter-list]) .

Examples:

```
transpose(a, n, m)
bisect(fct, -1.0, +1.0, x)
```

NOTE — The fourth example is not for level 0.

6.8.2.4 Goto-statements

A goto-statement shall indicate that further processing is to continue at the program-point denoted by the label in the goto-statement and shall cause the termination of all activations except

a) the activation containing the program-point; and

b) any activation containing the activation-point of an activation required by exceptions a) or b) not to be terminated.

c) a goto-statement shall be permitted to appear only within a function-block or a procedure-block; and

d) each occurrence of a goto-statement shall indicate that further processing shall continue at the program-point denoted by a label that occurs immediately before an empty statement immediately before the end word-symbol of the function-block or procedure-block closest containing the goto-statement.

goto-statement = "goto" label .

6.8.3 Structured-statements

6.8.3.1 General

structured-statement = compound-statement | conditional-statement
| repetitive-statement | with-statement .

statement-sequence = statement { ";" statement } .

The execution of a statement-sequence shall specify the execution of the statements of the statement-sequence in textual order, except as modified by execution of a goto-statement.

6.8.3.2 Compound-statements

A compound-statement shall specify execution of the statement-sequence of the compound-statement.

compound-statement = "begin" statement-sequence "end" .

Example:

```
    begin z := x; x := y; y := z end
```

6.8.3.3 Conditional-statements

conditional-statement = if-statement | case-statement .

6.8.3.4 If-statements

if-statement = "if" Boolean-expression "then" statement [else-part] .

else-part = "else" statement .

If the Boolean-expression of the if-statement yields the value true, the statement of the if-statement shall be executed. If the Boolean-expression yields the value false, the statement of the if-statement shall not be executed, and the statement of the else-part, if any, shall be executed.

An if-statement without an else-part shall not be immediately followed by the token else.

NOTE — An else-part is thus paired with the nearest preceding otherwise unpaired then.

Examples:

```
    if x < 1.5 then z := x + y else z := 1.5

    if p1 <> nil then p1 := p1↑.father
```

6.8.3.5 Case-statements

The values denoted by the case-constants of the case-constant-lists of the case-list-elements of a case-statement shall be distinct and of the same ordinal-type as the expression of the case-index of the case-statement. On execution of the case-statement the case-index shall be evaluated. That value shall then specify execution of the statement of the case-list-element closest-containing the case-constant denoting that value. One of the case-constants shall be equal to the value of the case-index upon entry to the case-statement; otherwise, it shall be an error.

NOTE — Case-constants are not the same as statement labels.

```
case-statement =  "case" case-index "of" case-list-element
                  { ";" case-list-element } [ ";" ] "end" .

case-list-element = case-constant-list ":" statement .

case-index = expression .
```

Example:

```
case operator of
  plus:    x := x + y;
  minus:   x := x - y;
  times:   x := x * y
end
```

6.8.3.6 Repetitive-statements

Repetitive-statements shall specify that certain statements are to be executed repeatedly.

```
repetitive-statement = repeat-statement | while-statement | for-statement .
```

6.8.3.7 Repeat-statements

```
repeat-statement = "repeat" statement-sequence "until" Boolean-expression .
```

The statement-sequence of the repeat-statement shall be repeatedly executed, except as modified by the execution of a goto-statement, until the Boolean-expression of the repeat-statement yields the value true on completion of the statement-sequence. The statement-sequence shall be executed at least once, because the Boolean-expression is evaluated after execution of the statement-sequence.

Example:

```
repeat
 k := i mod j;
 i := j;
 j := k
until j = 0
```

6.8.3.8 While-statements

```
while-statement = "while" Boolean-expression "do" statement .
```

The while-statement

```
    while b do body
```

shall be equivalent to

```
  begin
    if b then
    repeat
       body
    until not (b)
  end
```

Example:

```
      while i > 0 do
        begin if odd(i) then z := z * x;
        i := i div 2;
        x := sqr(x)
        end
```

6.8.3.9 For-statements

The for-statement shall specify that the statement of the for-statement is to be repeatedly executed while a progression of values is attributed to a variable denoted by the control-variable of the for-statement.

```
for-statement  =  "for" control-variable ":=" initial-value ( "to" | "downto" ) final-value
                  "do" statement .

control-variable  =  entire-variable .

initial-value  =  expression .

final-value  =  expression .
```

The control-variable shall be an entire-variable whose identifier is declared in the variable-declaration-part of the block closest-containing the for-statement. The control-variable shall possess an ordinal-type, and the initial-value and final-value shall be of a type compatible with this type. The initial-value and the final-value shall be assignment-compatible with the type possessed by the control-variable if the statement of the for-statement is executed. After a for-statement is executed, other than being left by a goto-statement, the control-variable shall be undefined. Neither a for-statement nor any procedure-and-function-declaration-part of the block that closest-contains a for-statement shall contain a statement threatening the variable denoted by the control-variable of the for-statement.

A statement S shall be designated as *threatening* a variable V if one or more of the following statements is true.

a) S is an assignment-statement and V is denoted by the variable-access of S.

b) S contains an actual variable parameter that denotes V.

c) S is a for-statement and the control-variable of S denotes V.

d) S is a statement specified using an equivalent program fragment containing a statement threatening V.

Apart from the restrictions imposed by these requirements, the for-statement

```
   for v := e1 to e2 do body
```

shall be equivalent to

```
begin
temp1 := e1;
temp2 := e2;
if temp1 <= temp2 then
  begin
  v := temp1;
  body;
  while v <> temp2 do
     begin
     v := succ(v);
     body
     end
  end
end
```

and the for-statement

```
for v := e1 downto e2 do body
```

shall be equivalent to

```
begin
temp1 := e1;
temp2 := e2;
if temp1 >= temp2 then
  begin
  v := temp1;
  body;
  while v <> temp2 do
    begin
    v := pred(v);
    body
    end
  end
end
```

where temp1 and temp2 denote auxiliary variables that the program does not otherwise contain, and that possess the type possessed by the variable v if that type is not a subrange-type; otherwise the host-type of the type possessed by the variable v.

Examples:

```
for i := 2 to 63 do
  if a[i] > max then max := a[i]

for i := 1 to 10 do
for j := 1 to 10 do
 begin
 x := 0;
 for k := 1 to 10 do
```

```
  x := x + m1[i,k] * m2[k,j];
 m[i,j] := x
 end

for i := 1 to 10 do
 for j := 1 to i - 1 do
  m[i][j] := 0.0

for c := blue downto red do
 q(c)
```

6.8.3.10 With-statements

with-statement = "with" record-variable-list "do" statement .

record-variable-list = record-variable { "," record-variable } .

field-designator-identifier = identifier .

A with-statement shall specify the execution of the statement of the with-statement. The occurrence of a record-variable as the only record-variable in the record-variable-list of a with-statement shall constitute the defining-point of each of the field-identifiers associated with components of the record-type possessed by the record-variable as a field-designator-identifier for the region that is the statement of the with-statement; each applied occurrence of a field-designator-identifier shall denote that component of the record-variable that is associated with the field-identifier by the record-type. The record-variable shall be accessed before the statement of the with-statement is executed, and that access shall establish a reference to the variable during the entire execution of the statement of the with-statement.

The statement

```
with v1,v2,...,vn do s
```

shall be equivalent to

```
with v1 do
  with v2 do
   ...
    with vn do s
```

Example:

```
with date do
 if month = 12 then
  begin month := 1; year := year + 1
  end
 else month := month+1
```

has the same effect on the variable date as

```
if date.month = 12 then
 begin date.month := 1; date.year := date.year+1
 end
else date.month := date.month+1
```

ANNEX D

(informative)

Estelle Tutorial

D.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 of the layers of Open Systems Interconnection (OSI) architecture defined by ISO.

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

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

Estelle can be briefly described as a technique that is based on an extended state transition model, i.e., a model of a non deterministic communicating automaton extended by the addition of Pascal language. More precisely, Estelle may be viewed as a set of extensions to ISO Pascal [42], level 0, which models a specified system as a hierarchical structure of communicating automata which:

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

Estelle permits a clear separation of the description of the communication interface between components of a specified system from the description of the internal behavior of each such component. As in Pascal, all manipulated objects are strongly typed. This property enables static detection (e.g. during compilation) of specification inconsistencies.

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

D.1.1 Modules and module instances

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

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

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

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

– interaction points;
– channels;
– interactions.

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

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

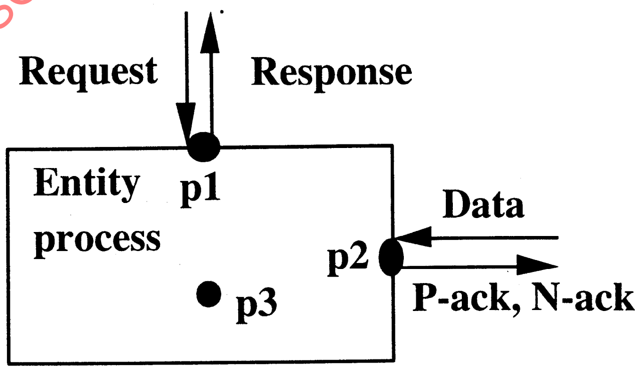


Figure D.1 - Graphical representation of a module

D.1.2 Structuring of modules

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

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

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

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

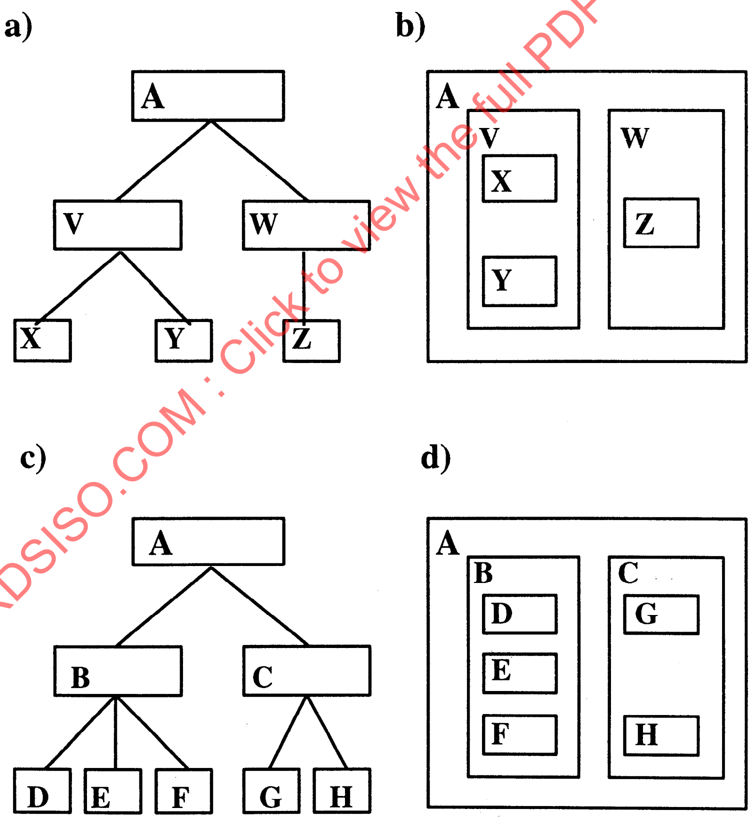


Figure D.2 - Graphical representation of a hierarchy of modules ((a) and (b)) and of a hierarchy of their instances ((c) and (d)).

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

D.1.3 Communication

Module instances within the hierarchy can communicate by

- message exchange;
- restricted sharing of variables.

D.1.3.1 Message exchange

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

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

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

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

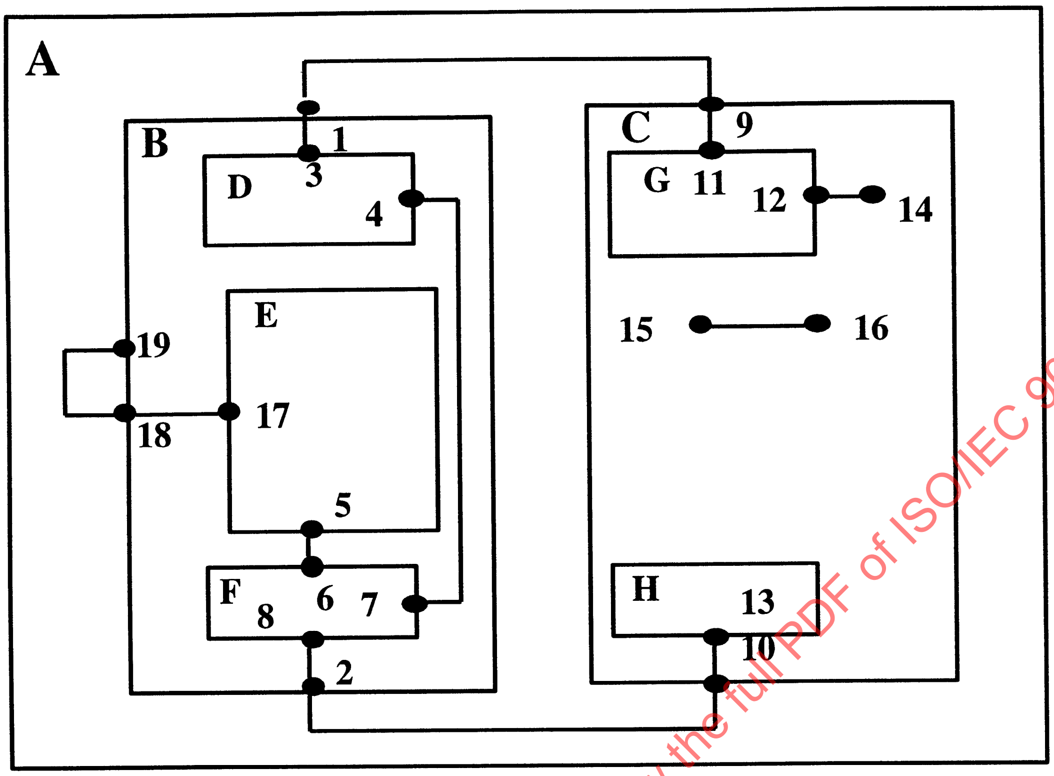


Figure D.3 - Module instances and their communication links.

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

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

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

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

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

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

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

D.1.3.2 Restricted sharing of variables

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

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

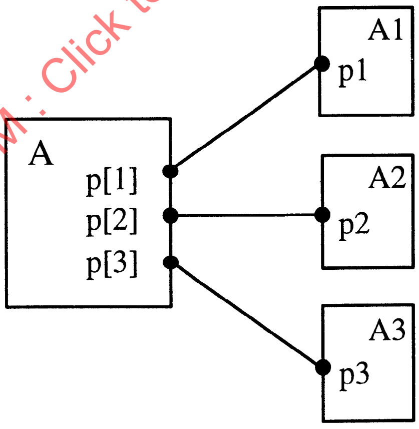


Figure D.4 - Multicast communication

D.1.4 Parallelism and Nondeterminism

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

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

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

or may be not attributed at all.

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

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

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

(a) Every active module (i.e., such that its definition includes at least one transition) shall be attributed,
(b) System modules shall not be nested within an attributed module,
(c) Modules attributed with **process** or **activity** shall be descendants of a system module,
(d) Modules attributed with **process** or **systemprocess** may be substructured only into modules attributed with either **process** or **activity**,
(e) Modules attributed with **activity** or **systemactivity** may be substructured only into modules attributed with **activity**.

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

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

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

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

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

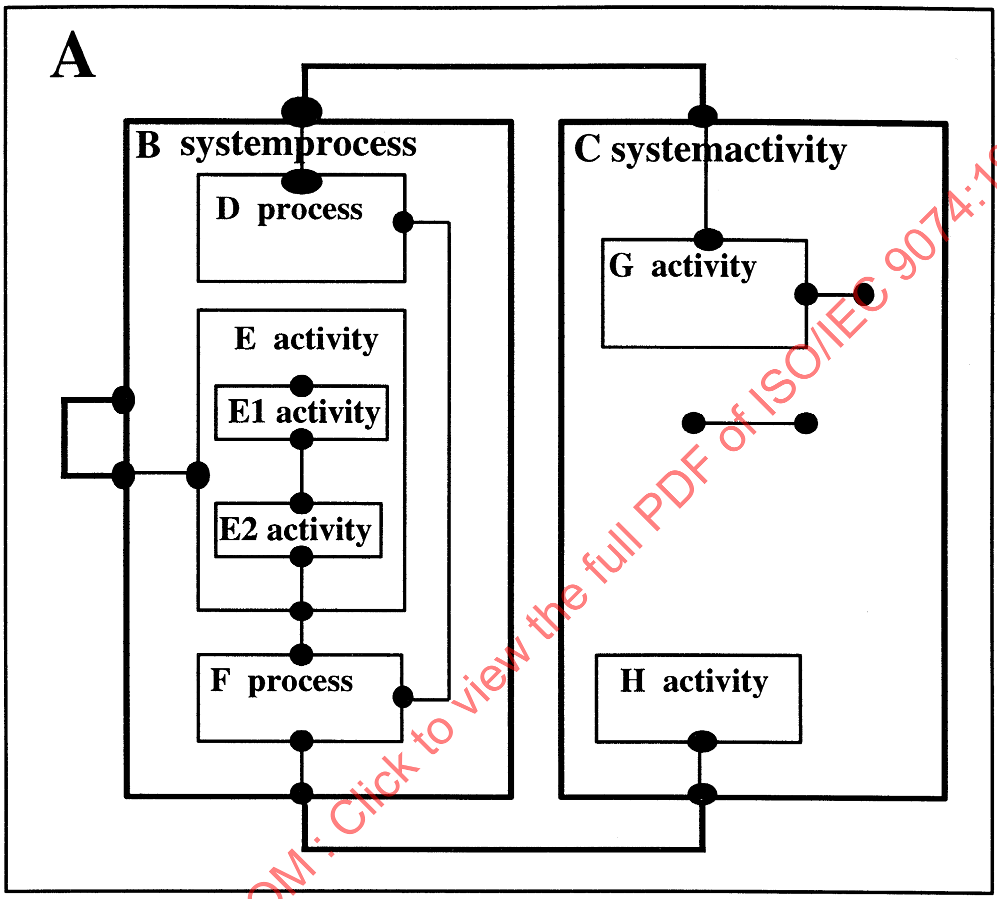


Figure D.5 - Attributed module instances and their communication links; different lines are used to represent system modules (bold), modules enclosing system modules (dotted) and others (non bold)

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

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

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

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

D.1.5 Dynamism within a system

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

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

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

D.1.6 Typing

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

D.1.7 Module internal behavior

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

An extended state is, in general, a complex structure composed of many components such as: value of the control state, values of variables, contents of FIFO queues associated with interaction points and the status of the module internal structure (submodule instances, bindings between interaction points, etc.). Initial states of a module instance are defined by an 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's state described above and may output interactions to the module environment. A compound-statements of Pascal is used to define the actions of a transitions .

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

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

D.1.8 Global behavior

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

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

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

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

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

D.2 Syntax and interpretation of Estelle concepts

D.2.1 Channels, interactions, and interaction points

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

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

```
by ROLE_Id2:   n1;
               n2;
               :
               nK;
by ROLE_Id1, ROLE_Id2:
               k1;
               k2;
               :
               kP;
```

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

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

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

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

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

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

Now, an interaction point p1 may be declared as follows

p1 : CHANNEL_Id(ROLE_Id1)

and another interaction point p2,

p2 : CHANNEL_Id(ROLE_Id2)

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

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

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

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

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

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

or

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

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

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

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

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

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

D.2.2 Modules

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

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

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

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

The following is an example of a module header definition:

```
module  A  systemactivity (R: boolean);
        ip     p  :   T(S) individual queue;
```

```
                    p2 :    W(K) common queue;
                    p3 :    U(S) common queue;
          export   X, Y : integer;  Z : boolean
     end;
```

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

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

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

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

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

In fact, at a conceptual level, two modules have been defined: one of which may be identified by the pair (A, B), and the second by the pair (A, C). The modules thus defined have the same external visibility (same interaction points p, p1, p2 and same exported variables X, Y, Z) and the same class attribute (systemactivity). But their 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 while refining the specification. The "external" bodies nicely serve to allow describing an overall system architecture without any detailed description of the system components. This feature is illustrated by the example in 4.1.

The body definition is composed of three parts:

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

D.2.3 Declaration part

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

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

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

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

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

```
module  A  (* see D.2.2 *) ··· end;
body B for A;
   ip  p1 : T1(R2) common queue; {internal ip}
   module  A1  activity  (k: integer);
       ip    p1 : T1(R1) individual queue;
             p2 : T1(R2) individual queue;
             p  : T(S)  individual queue;
   end;
   body B1 for A1; {body definition} end;
   body B2 for A1; {body definition} end;
end;
```

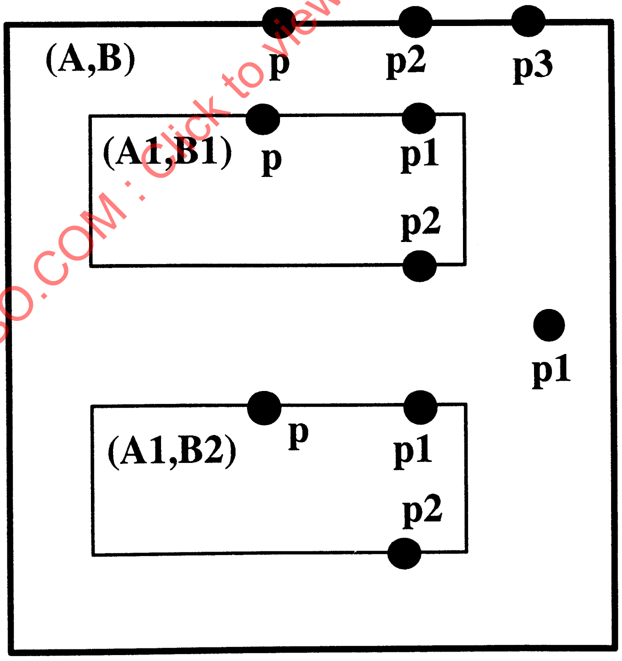


Figure D.6 - Textual hierarchy of modules

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

modvar X, Y, Z : A1

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

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

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

state IDLE, WAIT, OPEN, CLOSED

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

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

stateset IDWA = [IDLE, WAIT]

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

D.2.4 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 modules can be created. Creation of children modules during initialization defines their *initial architecture*

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

to IDLE

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

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

The above 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 with module type A1). The module instances (referenced by) X and Z are both instances of the same module (A1, B1) and module instance (referenced by) Y is an instance of the module (A1, B2). The module instances X, Y and Z have been initialized with different values (respectively 0, 1 and 4) of the parameter k declared within the header of the module A1. The concrete hierarchy of module instances of figure D.7 corresponds to the hierarchical pattern of module definitions from figure D.6.

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

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

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

        provided not R
            to WAIT
                begin
                    T := 8;
                    init X with B1 (0);
                    init Y with B2 (1);
        **connect** X.p1 **to** Y.p2;
        **attach** p **to** X.p
    **end**;
```

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

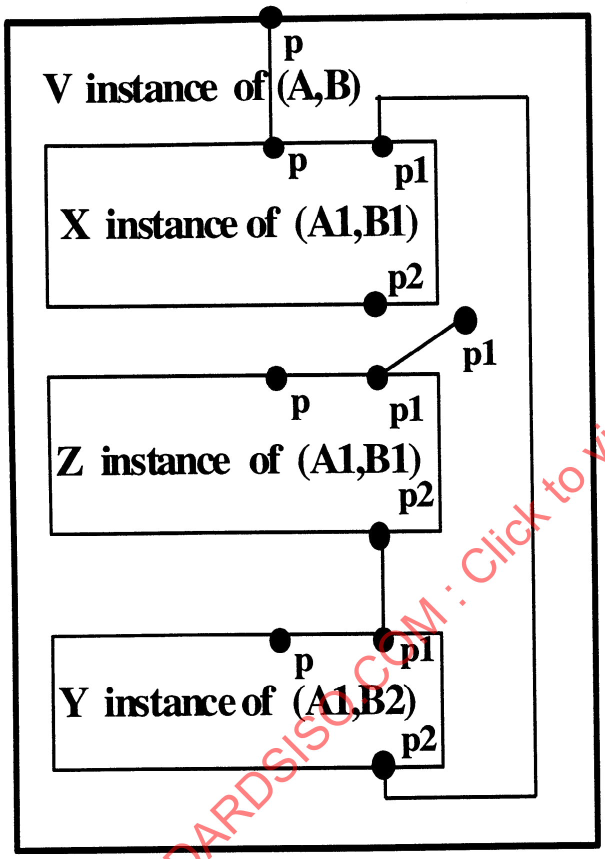


Figure D.7 - Module instance hierarchy corresponding to the textual pattern in figure D.6.

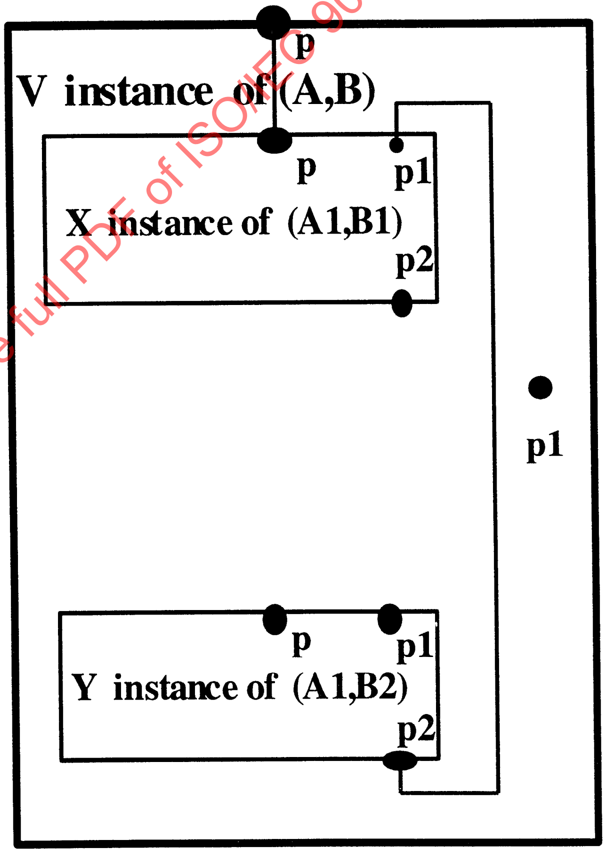


Figure D.8 - Module instance hierarchy corresponding to the textual pattern in figure D.6.

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

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

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

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

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

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

D.2.5 Transition part

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

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

Each expanded transition declaration is composed of two parts :

- clause-group;
- transition-block.

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

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

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