

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

ICS 35.100.01; 35.100.05

National foreword

This British Standard reproduces verbatim ISO/IEC 9074:1997 and implements it as the UK national standard. It supersedes BS ISO 9074:1989 which is withdrawn.

The UK participation in its preparation was entrusted to Technical Committee IST/21, Open Systems Interconnection, Data Management and Open Distributed Processing, which has the responsibility to:

- aid enquirers to understand the text;
- present to the responsible international/European committee any enquiries on the interpretation, or proposals for change, and keep the UK interests informed;
- monitor related international and European developments and promulgate them in the UK.

A list of organizations represented on this committee can be obtained on request to its secretary.

Cross-references

The British Standards which implement international or European publications referred to in this document may be found in the BSI Standards Catalogue under the section entitled “International Standards Correspondence Index”, or by using the “Find” facility of the BSI Standards Electronic Catalogue.

A British Standard does not purport to include all the necessary provisions of a contract. Users of British Standards are responsible for their correct application.

Compliance with a British Standard does not of itself confer immunity from legal obligations.

Summary of pages

This document comprises a front cover, an inside front cover, the ISO/IEC title page, pages ii to viii, pages 1 to 241 and a back cover.

This British Standard, having been prepared under the direction of the DISC Board, was published under the authority of the Standards Board and comes into effect on 15 April 1998

© BSI 1998

ISBN 0 580 29695 4

Amendments issued since publication

Amd. No.	Date	Text affected

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*



Contents

1 Scope 1

2 Normative References 1

3 Definitions 1

4 Conformance 2

5 The model 2

5.1 Module instance 3

5.2 Nesting of modules and parallelism 4

5.2.1 Nesting of modules 4

5.2.2 Parallelism and nondeterminism 8

5.3 Specification behavior 8

5.3.1 Local situations 8

5.3.2 Global instantaneous description of a module instance 9

5.3.3 Transitions selected for execution 9

5.3.4 Global behavior 10

5.3.5 Concept of time in the model 12

6 Definitional Conventions 13

6.1 Syntax definitions 13

6.2 Semantic notations 14

7 Language elements 16

7.1 Introduction 16

7.1.1 Character set 16

7.1.2 Estelle scope rules 16

7.2 Structure of a specification 18

7.2.1 Syntax 18

7.2.2 Constraints 19

7.2.3 Interpretation rules 19

7.2.4 Informal semantics 20

7.3 Declaration part 20

7.3.1 Syntax 20

7.3.2 Constraints 21

7.3.3 Informal semantics 21

7.3.4 Channel definition 21

7.3.5	Interaction points	23
7.3.6	Module header	25
7.3.7	Module body definition	27
7.3.8	Internal interaction points	28
7.3.9	Module variable declaration part	28
7.3.10	State definition part	29
7.3.11	State set definition part	30
7.4	References to Estelle objects	31
7.4.1	Module variable reference	31
7.4.2	Interaction point reference	31
7.4.3	Exported variable reference	32
7.5	Transition declarations	33
7.5.1	General introduction	33
7.5.2	Transition	33
7.5.3	To clause	38
7.5.4	From clause	39
7.5.5	Provided clause	39
7.5.6	When clause	40
7.5.7	Delay clause	41
7.5.8	Priority clause	42
7.5.9	Any clause	43
7.5.10	Initialization part	43
7.6	Estelle statements	44
7.6.1	Module instance creation	44
7.6.2	Release and termination of module instances	45
7.6.3	Connect operation	46
7.6.4	Attach operation	47
7.6.5	Disconnect operation	48
7.6.6	Detach operation	48
7.6.7	Summary of binding operations	49
7.6.8	Output statement	50
7.6.9	All statement	53
7.6.10	Forone statement	55
7.6.11	Exist expression	56
7.7	Reserved words	56
7.7.1	Syntax	57
7.7.2	Constraints	57
8	Extensions and restrictions to ISO Pascal	57
8.1	Simple changes to Pascal syntax	57
8.1.1	Syntax	57
8.2	Extensions	58
8.2.1	Integers and real numbers	58
8.2.2	Functions and procedures	59
8.2.3	Implementation defined elements	59
8.2.4	Directives	60
8.2.5	Pure procedures and functions	61
8.2.6	Expression	62

8.2.7	Assignment operation	63
8.3	Restrictions	63
8.3.1	Errors	64
8.3.2	File manipulation	64
8.3.3	Label declarations and goto statements	64
8.3.4	Program statement	64
8.3.5	Expressions and functions	64
9	Semantics of Estelle constructs	65
9.1	General scheme of definitions	65
9.2	Primitives and organization of the clause	66
9.3	External context environment and channel definition	67
9.3.1	Interactions	69
9.3.2	Channel definition interpretation	69
9.4	Module instances	70
9.4.1	Identifier categories of a module instance	70
9.4.2	Internal extensions of context environment and recursive assumptions	72
9.4.3	Interaction points of a module instance and related notions	72
9.4.4	Locations, variable allocations and variable visibility within a module instance	74
9.4.5	States of a module instance	75
9.5	Transitions of a module instance	79
9.5.1	Transition statement	80
9.5.2	Transition interpretation	81
9.5.3	Linked interaction points	82
9.5.4	Extension of the transition interpretation over gids	83
9.6	Interpretation of specific Estelle constructs	84
9.6.1	External references	84
9.6.2	Semantics of transition clauses	86
9.6.3	Enabled transitions and initial states	88
9.6.4	Delay values and time interpretation in Estelle	89
9.6.5	Fireable transitions and offered transitions	92
9.6.6	Semantics of primitive Estelle statements	93
Annexes		103
A	Collected syntax	103
A.1	Collected syntax from clause 7	103
A.2	Collected syntax from clause 8	111
A.3	Collected syntax from annex C	112
B	User guidelines	124
B.1	User data management	124
B.1.1	Purpose of user data management	124
B.1.2	Principles	124
B.1.3	Encode procedure	124
B.1.4	Decode procedure	125
B.1.5	Guidelines	125
B.2	Alternating bit example	130

C	Pascal subset used by Estelle	138
D	Estelle Tutorial	176
D.0	Introduction	176
D.1	A brief overview of the principal concepts in Estelle	177
D.1.1	Modules and module instances	177
D.1.2	Structuring of modules	178
D.1.3	Communication	179
D.1.4	Parallelism and Nondeterminism	181
D.1.5	Dynamism within a system	184
D.1.6	Typing	184
D.1.7	Module internal behavior	184
D.1.8	Global behavior	185
D.2	Syntax and interpretation of Estelle concepts	185
D.2.1	Channels, interactions, and interaction points	185
D.2.2	Modules	187
D.2.3	Declaration part	188
D.2.4	Initialization part	190
D.2.5	Transition part	193
D.2.6	Shorthands	195
D.3	Estelle statements	198
D.3.1	Init statement	198
D.3.2	Connect statement	199
D.3.3	Disconnect statement	200
D.3.4	Attach statement	201
D.3.5	Detach statement	203
D.3.6	Release and terminate statements	203
D.3.7	Output statement	204
D.3.8	All and forone statements, exist expression	205
D.3.9	Pascal restrictions in Estelle	208
D.4	Behavior of Estelle specifications	209
D.4.1	Specification module	209
D.4.2	Internal behavior of a module	212
D.4.3	Global behavior semantics	218
D.4.4	Comparison of behaviors using a common example	223
D.5	References	228
	Indices	233

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.

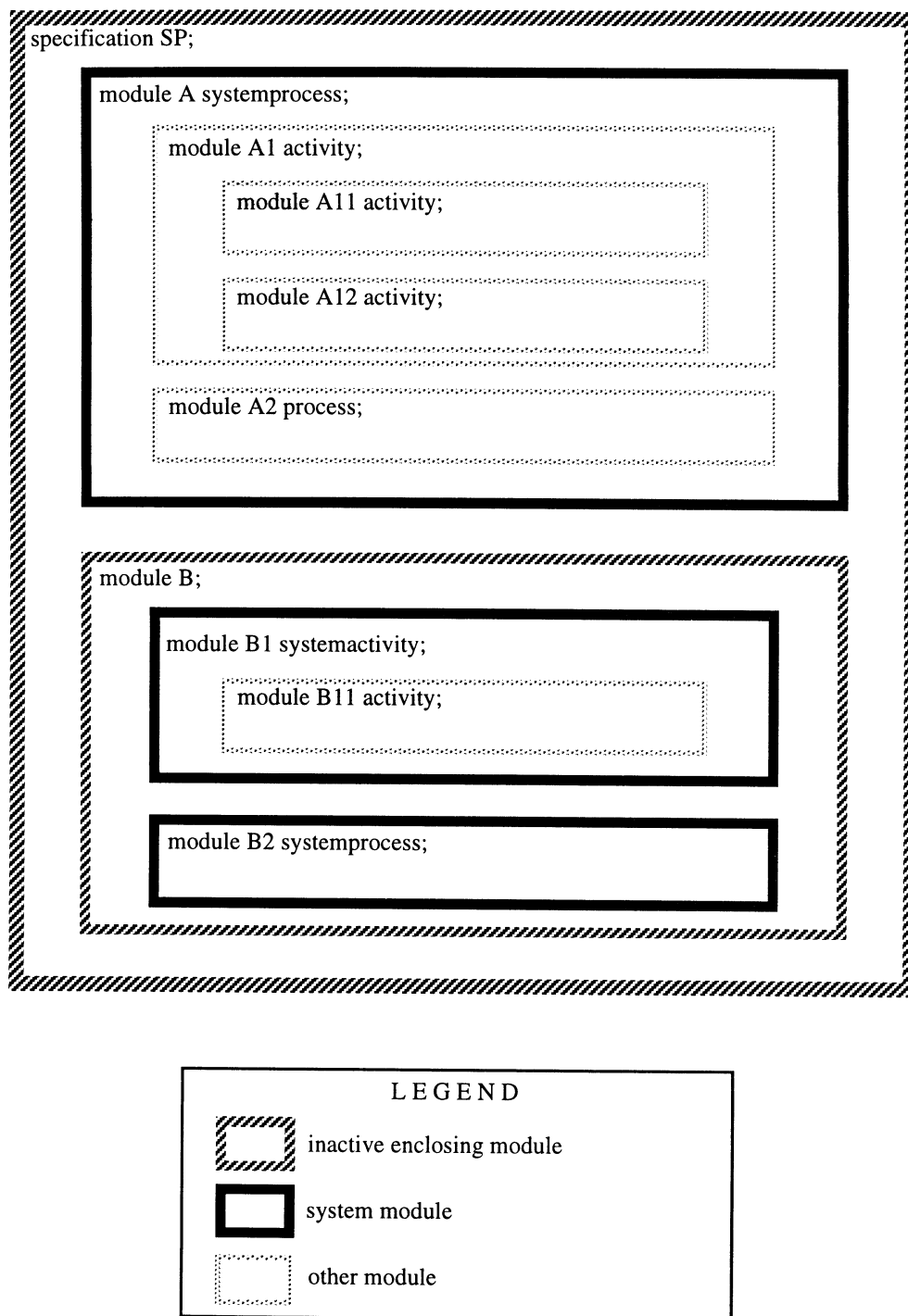


Figure 1(a)

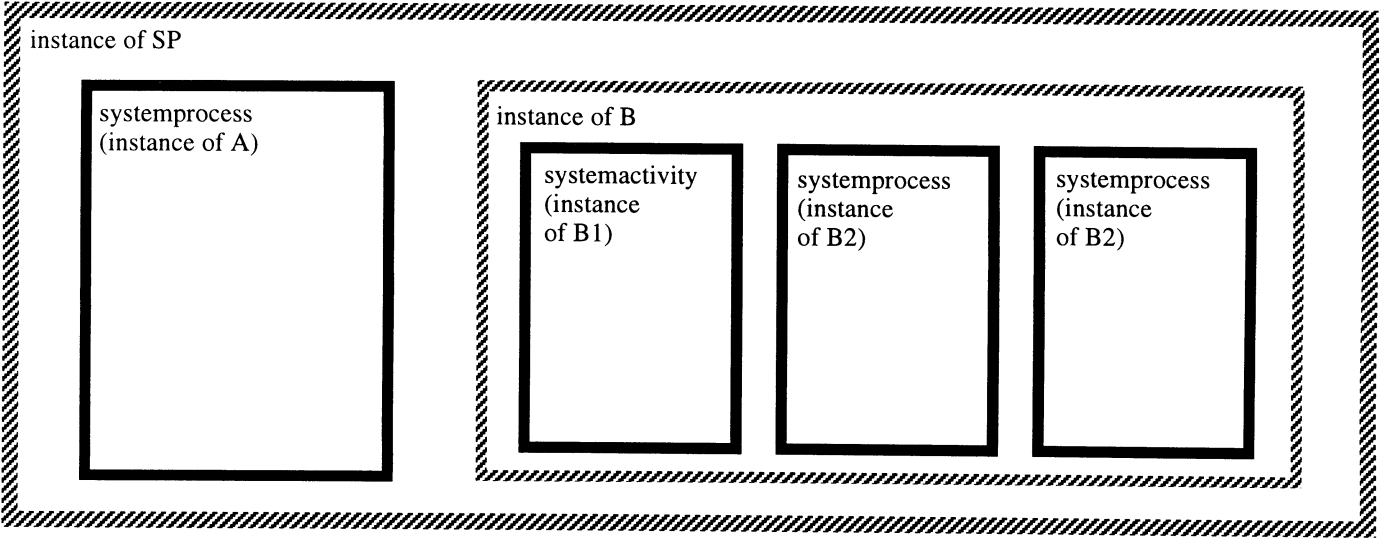


Figure 1(b)

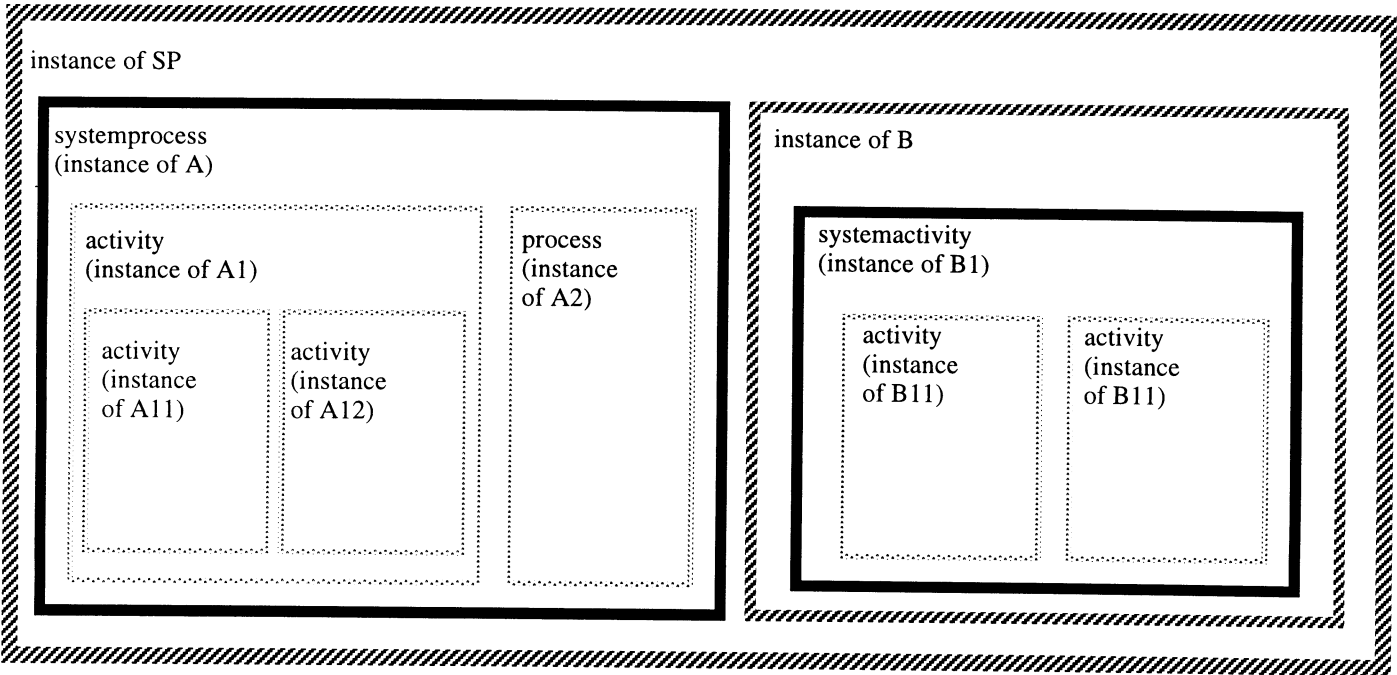


Figure 1(c)

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 $\{\text{null}\}$ is identified with the empty set).
- (b) Let $P_1, P_2, \dots, P_k$, $k \geq 1$, 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 $t_P \neq \text{null}$, then $AS(gid_P) = \{t_P\}$ (parent priority).

(2) If $t_p = \text{null}$ 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 $AS(gid_{p_i})$, $i = 1, \dots, k$, 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 $t_p = \text{null}$ 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 $S_1, \dots, S_n$, 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 ($i = 1, \dots, n$) 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: $(gid_{SP}; A_1, \dots, A_n)$, 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 $i = 1, \dots, n$.

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

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

- (a) if $A_i = \emptyset$, then for every $AS(gid_{SP}/S_i) \in AS^*(gid_{SP}/S_i)$,
 $(gid_{SP}; A_1, \dots, AS(gid_{SP}/S_i), \dots, A_n)$ is a next global situation of sit .

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

$(t(\text{gid}_{\text{SP}}); A_1, \dots, A_i \setminus \{t\}, \dots, A_n)$ 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 $\text{AS}(\text{gid}_{\text{SP}}/S_i)$ 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 , $\text{sit}_0, \text{sit}_1, \dots, \text{sit}_j, \dots$, 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 ($i = 1, \dots, n$) 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
$x \psi y$ $a_1 \psi \dots \psi a_n$	$xy \mid yx$ 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 metasympols.

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
- $\text{POWER}(A)$ — 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 — $\text{head}(L) = a$ if $L = aL'$, i.e., **head** is the function defined for nonempty lists that returns the first element of the list L

tail — $\text{tail}(L) = L'$ 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

— $\text{append}(L', L) = LL'$, 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)

$\text{dom}(f)$

— 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:

$\text{dom}(f[g]) = \text{dom}(f) \cup \text{dom}(g)$, and

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

$f(a_1/v_1, \dots, a_n/v_n)$

— simultaneous substitution, i.e., an operator that, for a given function $f : A \rightsquigarrow B$, $a_1, \dots, a_n \in \text{dom}(f)$ and $v_1, \dots, v_n \in B$, returns the function of the same domain and identically defined as f for all $a \neq a_i$ ($i = 1, \dots, n$), 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** $x \in \text{Set}$ **do** **meta-stm**

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

Compound meta-statement: (**meta-stm**₁; **meta-stm**₂; ...; **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: $S = (s.\text{ie}, s.\text{Loc}, \dots, s.\text{out})$. An assignment to a state component, e.g., $s.\text{Loc} := f(s.\text{Loc}, \dots)$ describes a state transformation which for a given state s returns a new state that is identical to s except that $s.\text{Loc}$ 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); by R1 : m1; by R2 : m2; by R1, R2 : m3; by R2 : m4; by R1, R2 : m5;	is equivalent to	channel H(R1,R2); by R1 : m1; by R2 : m2; m4; by R1, R2 : m3; 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 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

$$\begin{aligned} \text{module-variable} = & \text{ module-variable-identifier} \\ & | \text{ module-variable-identifier } "[\text{ INDEX-EXPRESSION} \\ & \quad \{ \text{ " , " INDEX-EXPRESSION } \} "]" . \end{aligned}$$

$$\text{module-variable-identifier} = \text{ IDENTIFIER } .$$

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

$$\text{connect-ip} = \text{ child-external-ip } | \text{ internal-ip } .$$

$$\text{child-external-ip} = \text{ module-variable } "." \text{ external-ip } .$$

$$\text{external-ip} = \text{ interaction-point-reference } .$$

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
  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 $c_1, \dots, c_n$ ($n \geq 1$) are clauses of the same category C and $t_1, \dots, t_n$ are well-formed **transition-groups**, then $c_1 t_1 \dots c_n t_n$ 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 $t_1, \dots, t_n$;
 - (2) if C is a when-clause category then a delay-clause does not appear in $t_1, \dots, t_n$;
 - (3) if C is a delay-clause category then a when-clause does not appear in $t_1, \dots, t_n$.

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 $i = 1, \dots, n$.

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

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

$\text{insert}(c, \text{“trans” } t_1 \cdots \text{“trans” } t_n) = \text{“trans” } c t_1 \text{ “trans” } c t_2 \cdots \text{“trans” } c t_n$

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

if t is expanded:
 $\text{expand}'(\text{“trans” } t) = \text{“trans” } t$
 otherwise:
 recall that $t = c_1 t_1 \cdots c_n t_n$.
 There are two cases:
 if $n = 1$:
 $\text{expand}'(\text{“trans” } t) = \text{insert}(c_1, \text{expand}'(\text{“trans” } \text{replace}(t_1)))$
 if $n > 1$:
 $\text{expand}'(\text{“trans” } t) = \text{insert}(c_1, \text{expand}'(\text{“trans” } \text{replace}(t_1)))$
 $\text{expand}'(\text{“trans” } c_2 t_2 \cdots c_n t_n)$

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

$\text{expand}(\text{“trans” } t) = \text{expand}'(\text{“trans” } \text{replace}(t))$

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-declarations** that contain no **any-clause**. The transitions in this sequence are called *simple*. Each of these **transition-declarations** 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 $c_1t_1 \cdots c_nt_n$ (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 p1  t1
provided p2  t2
...
provided otherwise tn
```

is semantically equivalent to

```
provided p1  t1
provided p2  t2
...
provided NOT (p1 OR p2 OR ... OR pn-1)  tn.
```

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).