INTERNATIONAL TELECOMMUNICATION UNION

CCITT

THE INTERNATIONAL
TELEGRAPH AND TELEPHONE
CONSULTATIVE COMMITTEE

BLUE BOOK

VOLUME X – FASCICLE X.1

FUNCTIONAL SPECIFICATION AND DESCRIPTION LANGUAGE (SDL)

CRITERIA FOR USING FORMAL DESCRIPTION TECHNIQUES (FDTs)

**RECOMMENDATION Z.100 AND ANNEXES A, B, C AND E,
RECOMMENDATION Z.110**

IXTH PLENARY ASSEMBLY
MELBOURNE, 14-25 NOVEMBER 1988

Geneva 1989

INTERNATIONAL TELECOMMUNICATION UNION

CCITT

THE INTERNATIONAL
TELEGRAPH AND TELEPHONE
CONSULTATIVE COMMITTEE

BLUE BOOK

VOLUME X – FASCICLE X.1

FUNCTIONAL SPECIFICATION AND DESCRIPTION LANGUAGE (SDL)

CRITERIA FOR USING FORMAL DESCRIPTION TECHNIQUES (FDTs)

**RECOMMENDATION Z.100 AND ANNEXES A, B, C AND E,
RECOMMENDATION Z.110**

IXTH PLENARY ASSEMBLY
MELBOURNE, 14-25 NOVEMBER 1988

Geneva 1989

ISBN 92-61-03751-8

Printed in Switzerland

CONTENTS OF THE CCITT BOOK
APPLICABLE AFTER THE NINTH PLENARY ASSEMBLY (1988)

BLUE BOOK

FASCICLE III.6 – Line transmission of non-telephone signals. Transmission of sound-programme and television signals. Series H and J Recommendations (Study Group XV).

FASCICLE III.7 – Integrated Services Digital Network (ISDN) – General structure and service capabilities. Recommendations I.110-I.257 (Study Group XVIII).

FASCICLE III.8 – Integrated Services Digital Network (ISDN) – Overall network aspects and functions, ISDN user-network interfaces. Recommendations I.310-I.470 (Study Group XVIII).

FASCICLE III.9 – Integrated Services Digital Network (ISDN) – Internetwork interfaces and maintenance principles. Recommendations I.500-I.605 (Study Group XVIII).

Volume IV

FASCICLE IV.1 – General maintenance principles: maintenance of international transmission systems and telephone circuits. Recommendations M.10-M.782 (Study Group IV).

FASCICLE IV.2 – Maintenance of international telegraph, phototelegraph and leased circuits. Maintenance of the international public telephone network. Maintenance of maritime satellite and data transmission systems. Recommendations M.800-M.1375 (Study Group IV).

FASCICLE IV.3 – Maintenance of international sound-programme and television transmission circuits. Series N Recommendations (Study Group IV).

FASCICLE IV.4 – Specifications for measuring equipment. Series O Recommendations (Study Group IV).

Volume V – Telephone transmission quality. Series P Recommendations (Study Group XII).

Volume VI

FASCICLE VI.1 – General Recommendations on telephone switching and signalling. Functions and information flows for services in the ISDN. Supplements. Recommendations Q.1-Q.118 *bis* (Study Group XI).

FASCICLE VI.2 – Specifications of Signalling Systems Nos. 4 and 5. Recommendations Q.120-Q.180 (Study Group XI).

FASCICLE VI.3 – Specifications of Signalling System No. 6. Recommendations Q.251-Q.300 (Study Group XI).

FASCICLE VI.4 – Specifications of Signalling Systems R1 and R2. Recommendations Q.310-Q.490 (Study Group XI).

FASCICLE VI.5 – Digital local, transit, combined and international exchanges in integrated digital networks and mixed analogue-digital networks. Supplements. Recommendations Q.500-Q.554 (Study Group XI).

FASCICLE VI.6 – Interworking of signalling systems. Recommendations Q.601-Q.699 (Study Group XI).

FASCICLE VI.7 – Specifications of Signalling System No. 7. Recommendations Q.700-Q.716 (Study Group XI).

FASCICLE VI.8 – Specifications of Signalling System No. 7. Recommendations Q.721-Q.766 (Study Group XI).

FASCICLE VI.9 – Specifications of Signalling System No. 7. Recommendations Q.771-Q.795 (Study Group XI).

FASCICLE VI.10 – Digital subscriber signalling system No. 1 (DSS 1), data link layer. Recommendations Q.920-Q.921 (Study Group XI).

FASCICLE VI.11 – Digital subscriber signalling system No. 1 (DSS 1), network layer, user-network management. Recommendations Q.930-Q.940 (Study Group XI).

FASCICLE VI.12 – Public land mobile network. Interworking with ISDN and PSTN. Recommendations Q.1000-Q.1032 (Study Group XI).

FASCICLE VI.13 – Public land mobile network. Mobile application part and interfaces. Recommendations Q.1051-Q.1063 (Study Group XI).

FASCICLE VI.14 – Interworking with satellite mobile systems. Recommendations Q.1100-Q.1152 (Study Group XI).

Volume VII

FASCICLE VII.1 – Telegraph transmission. Series R Recommendations. Telegraph services terminal equipment. Series S Recommendations (Study Group IX).

FASCICLE VII.2 – Telegraph switching. Series U Recommendations (Study Group IX).

FASCICLE VII.3 – Terminal equipment and protocols for telematic services. Recommendations T.0-T.63 (Study Group VIII).

FASCICLE VII.4 – Conformance testing procedures for the Teletex Recommendations. Recommendation T.64 (Study Group VIII).

FASCICLE VII.5 – Terminal equipment and protocols for telematic services. Recommendations T.65-T.101, T.150-T.390 (Study Group VIII).

FASCICLE VII.6 – Terminal equipment and protocols for telematic services. Recommendations T.400-T.418 (Study Group VIII).

FASCICLE VII.7 – Terminal equipment and protocols for telematic services. Recommendations T.431-T.564 (Study Group VIII).

Volume VIII

FASCICLE VIII.1 – Data communication over the telephone network. Series V Recommendations (Study Group XVII).

FASCICLE VIII.2 – Data communication networks: services and facilities, interfaces. Recommendations X.1-X.32 (Study Group VII).

FASCICLE VIII.3 – Data communication networks: transmission, signalling and switching, network aspects, maintenance and administrative arrangements. Recommendations X.40-X.181 (Study Group VII).

FASCICLE VIII.4 – Data communication networks: Open Systems Interconnection (OSI) – Model and notation, service definition. Recommendations X.200-X.219 (Study Group VII).

FASCICLE VIII.5 – Data communication networks: Open Systems Interconnection (OSI) – Protocol specifications, conformance testing. Recommendations X.220-X.290 (Study Group VII).

FASCICLE VIII.6 – Data communication networks: interworking between networks, mobile data transmission systems, internetwork management. Recommendations X.300-X.370 (Study Group VII).

FASCICLE VIII.7 – Data communication networks: message handling systems. Recommendations X.400-X.420 (Study Group VII).

FASCICLE VIII.8 – Data communication networks: directory. Recommendations X.500-X.521 (Study Group VII).

Volume IX – Protection against interference. Series K Recommendations (Study Group V). Construction, installation and protection of cable and other elements of outside plant. Series L Recommendations (Study Group VI).

Volume X

FASCICLE X.1 – Functional Specification and Description Language (SDL). Criteria for using Formal Description Techniques (FDTs). Recommendation Z.100 and Annexes A, B, C and E, Recommendation Z.110 (Study Group X).

FASCICLE X.2 – Annex D to Recommendation Z.100: SDL user guidelines (Study Group X).

FASCICLE X.3 – Annex F.1 to Recommendation Z.100: SDL formal definition. Introduction (Study Group X).

FASCICLE X.4 – Annex F.2 to Recommendation Z.100: SDL formal definition. Static semantics (Study Group X).

FASCICLE X.5 – Annex F.3 to Recommendation Z.100: SDL formal definition. Dynamic semantics (Study Group X).

FASCICLE X.6 – CCITT High Level Language (CHILL). Recommendation Z.200 (Study Group X).

FASCICLE X.7 – Man-Machine Language (MML). Recommendations Z.301-Z.341 (Study Group X).

TABLE OF CONTENTS OF FASCICLE X.1 OF THE BLUE BOOK

Recommendation Z.100 and Annexes A, B, C and E

Recommendation Z.110

Functional specification and description language (SDL)

Criteria for using formal description techniques (FDTs)

PRELIMINARY NOTES

1 The Questions entrusted to each Study Group for the Study Period 1989-1992 can be found in Contribution No. 1 to that Study Group.

2 In this Fascicle, the expression "Administration" is used for shortness to indicate both a telecommunication Administration and a recognized private operating agency.

FASCICLE X.1

Recommendation Z.100 and Annexes A, B, C and E
Recommendation Z.110

FUNCTIONAL SPECIFICATION AND DESCRIPTION LANGUAGE (SDL)

CRITERIA FOR USING FORMAL DESCRIPTION TECHNIQUES (FDTs)

PAGE INTENTIONALLY LEFT BLANK

PAGE LAISSEE EN BLANC INTENTIONNELLEMENT

TABLE OF CONTENTS OF Z.100

Recommendation Z.100

Specification and Description Language (SDL)

PRELIMINARY NOTE

This Recommendation replaces Recommendations Z.100 to Z.104 and Recommendation X.250 of the CCITT RED BOOK.

1 Introduction to SDL

1.1 *Introduction*

The purpose of recommending SDL (Specification and Description Language) is to provide a language for unambiguous specification and description of the behaviour of telecommunications systems. The specifications and descriptions using SDL are intended to be formal in the sense that it is possible to analyse and interpret them unambiguously.

The terms specification and description are used with the following meaning:

a) a specification of a system is the description of its required behaviour, and

b) a description of a system is the description of its actual behaviour.

Note - Since there is no distinction between use of SDL for specification and its use for description, the term specification is in the subsequent text used for both required behaviour and actual behaviour.

A system specification, in a broad sense, is the specification of both the behaviour and a set of general parameters of the system. However SDL aims only to specify the behavioural aspects of a system; the general parameters describing properties like capacity and weight have to be described using different techniques.

1.1.1 *Objectives*

The general objectives when defining SDL have been to provide a language that:

a) is easy to learn, use and interpret;

b) provides unambiguous specification for ordering and tendering;

c) may be extended to cover new developments;

d) is able to support several methodologies of system specification and design, without assuming any one of these.

1.1.2 *Applications*

The main area of application for SDL is the specification of the behaviour of aspects of real time systems. Applications include:

a) call processing (e.g. call handling, telephony signalling, metering) in switching systems;

b) maintenance and fault treatment (e.g. alarms, automatic fault clearance, routine tests) in general telecommunications systems;

c) system control (e.g. overload control, modification and extension procedures);

d) operation & maintenance functions, network management;

e) data communication protocols.

SDL can, of course, be used for the functional specification of the behaviour of any object whose behaviour can be specified using a discrete model; i.e. the object communicates with its environment by discrete messages.

SDL is a rich language and can be used for both high level informal (and/or formally incomplete) specifications, semi-formal and detailed specifications. The user must choose the appropriate parts of SDL for the intended level of communication and the environment in which the language is being used. Depending on the environment in which a specification is used then many aspects may be left to the common understanding between the source and the destination of the specification.

Thus SDL may be used for producing:

a) facility requirements,

b) system specifications,

c) CCITT Recommendations,

d) system design specifications,

e) detailed specifications,

f) system design (both high level and detailed),

g) system testing

and the user organization can choose the appropriate level of application of SDL.

1.1.3 *System specification*

An SDL specification defines a system behaviour in a stimulus/response fashion, assuming that both stimuli and responses are discrete and carry information. In particular a system specification is seen as the sequence of responses to any given sequence of stimuli.

The system specification model is based on the concept of communicating extended finite state machines.

SDL also provides structuring concepts which facilitate the specification of large and/or complex systems. These constructs allow the partitioning of the system specification into manageable units that may be handled and understood independently. Partitioning may be performed in a number of steps resulting in a hierarchical structure of units defining the system at different levels.

1.2 *SDL grammars*

SDL gives a choice of two different syntactic forms to use when representing a system; a Graphic Representation (SDL/GR), and a textual Phrase Representation (SDL/PR). As both are concrete representations of the same SDL semantics, they are equivalent. In particular they are both equivalent to an abstract grammar for the corresponding concepts.

A subset of SDL/PR is common with SDL/GR. This subset is called common textual grammar.

Figure 1.1 shows the relationships between SDL/PR, SDL/GR, the concrete grammars and the abstract grammar.

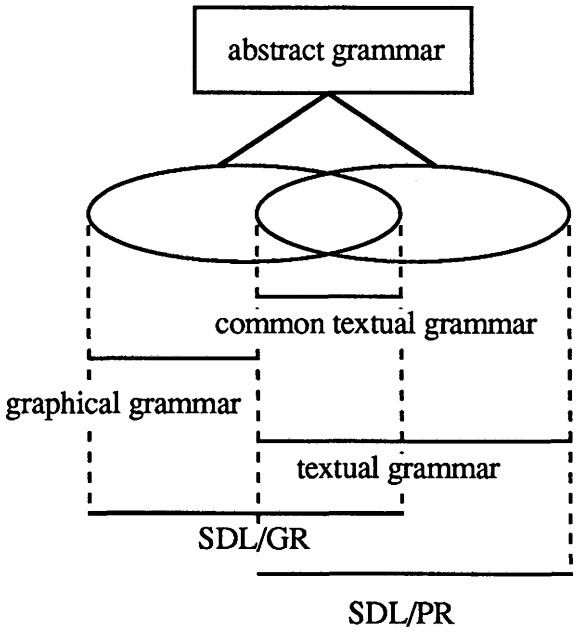


FIGURE 1.1
SDL grammars

Each of the concrete grammars has a definition of its own syntax and of its relationship to the abstract grammar (i.e. how to transform into the abstract syntax). Using this approach there is only one definition of the semantics of SDL; each of the concrete grammars will inherit the semantics via its relations to the abstract grammar. This approach also ensures that SDL/PR and SDL/GR are equivalent.

A formal definition of SDL is also provided which defines how to transform a system specification into the abstract syntax and define how to interpret a specification, given in terms of the abstract syntax.

1.3 *Basic definitions*

Some general concepts and conventions are used throughout this Recommendation, their definitions are given in the following:

1.3.1 *Type, definition and instance*

In the Recommendation, the concepts of type, type instance and their relationship are fundamental. The schema and terminology defined below and shown in Figure 1.2 are used.

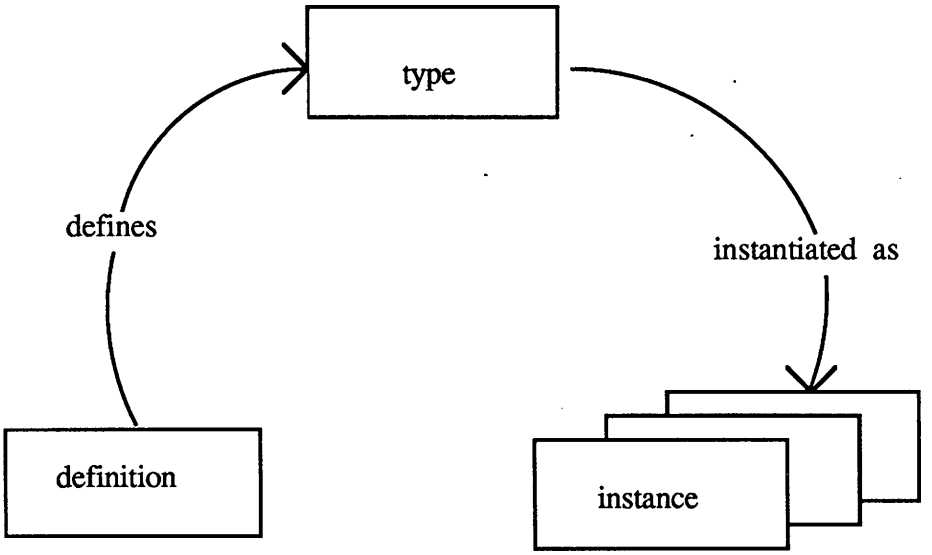


FIGURE 1.2

The type concept

Types are defined by means of definitions. A definition of a type defines all properties associated with that type. A type may be instantiated in any number of instances. Any instance of a particular type has all the properties defined for that type.

This schema applies to several SDL concepts, e.g. system definitions and system instances, process definitions and process instances.

Data type is a special class of type (see § 2.3 and § 5).

Note - To avoid cumbersome text, the convention is used that the term instance may be omitted. For example "a system is interpreted....." means "a system instance is interpreted....".

1.3.2 *Environment*

Systems specified in SDL behave according to the stimuli received from the external world. This external world is called the environment of the system being specified.

It is assumed that there are one or more process instances in the environment , and therefore signals flowing from the environment toward the system have associated identities of these process instances. These processes have PId values different from any PId value in the system (see § 5.6.10)

Although the behaviour of the environment is nondeterministic, it must obey the constraints given by the system specification.

1.3.3 *Errors*

A system specification is a valid SDL system specification only if it satisfies the syntactic rules and the static conditions of SDL.

If a valid SDL specification is interpreted and a dynamic condition is violated then an error occurs. An interpretation of a system specification which leads to an error means that the subsequent behaviour of the system cannot be derived from the specification.

1.4 *Presentation style*

1.4.1 *Division of text*

In § 2, 3, 4, 5 the Recommendation is organised by topics described by an optional introduction followed by titled enumeration items for:

a) *Abstract grammar* — described by abstract syntax and static conditions for well-formedness.

b) *Concrete textual grammar* — both the common textual grammar used for SDL/PR and SDL/GR and the grammar used only for SDL/PR. This grammar is described by the textual syntax, static conditions and well formedness rules for the textual syntax, and the relationship of the textual syntax with the abstract syntax.

c) *Concrete graphical grammar* — described by the graphical syntax, static conditions and well-formedness rules for the graphical syntax, the relationship of this syntax with the abstract syntax, and some additional drawing rules (to those in § 2.2.4).

d) *Semantics* — gives meaning to a type, provides the properties it has, the way in which an instance of that type is interpreted and any dynamic conditions which have to be fulfilled for the instance of that type to be well behaved in the SDL sense.

e) *Model* — gives the mapping for shorthand notations expressed in terms of previously defined strict concrete syntax constructs.

f) *Examples*

1.4.2 *Titled enumeration items*

Where a topic has an introduction followed by a titled enumeration item then the introduction is considered to be an informal part of the Recommendation presented only to aid understanding and not to make the Recommendation complete.

If there is no text for a titled enumeration item the whole item is omitted.

The remainder of this section describes the other special formalisms used in each titled enumeration item and the titles used. It can also be considered as an example of the typographical layout of first level titled enumeration items defined above where this text is part of an introductory section.

Abstract grammar

The abstract syntax notation is defined in § 1.5.1.

If the titled enumeration item *Abstract grammar* is omitted, then there is no additional abstract syntax for the topic being introduced and the concrete syntax will map onto the abstract syntax defined by another numbered text section.

The rules in the abstract syntax may be referred to from any of the titled enumeration items by use of the rule name in italics.

The rules in the formal notation may be followed by paragraphs which define conditions which must be satisfied by a well-formed SDL definition and which can be checked without interpretation of an instance. The static conditions at this point refer only to the abstract syntax. Static conditions which are only relevant for the concrete syntax are defined after the concrete syntax. Together with the abstract syntax the static conditions for the abstract syntax define the abstract grammar of the language.

Concrete textual grammar

The concrete textual syntax is specified in the extended Backus-Naur Form of syntax description defined in Recommendation Z.200 paragraph 2.1 (see also § 1.5.2).

The textual syntax is followed by paragraphs defining the static conditions which must be satisfied in a well-formed text and which can be checked without interpretation of an instance. Static conditions (if any) for the abstract grammar also apply.

In many cases there is a simple relationship between the concrete and abstract syntax as a concrete syntax rule is simply represented by a single rule in the abstract syntax. When the same name is used in the abstract and concrete syntax in order to signify that they represent the same concept, then the text "<x> in the concrete syntax represents *X* in the abstract syntax" is implied in the language description and is often omitted. In this context case is ignored but underlined semantic sub-categories are significant.

Concrete textual syntax which is not a shorthand form (derived syntax modelled by other SDL constructs) is strict concrete textual syntax. The relationship from concrete textual syntax to abstract syntax is defined only for the strict concrete textual syntax.

The relationship between concrete textual syntax and abstract syntax is omitted if the topic being defined is a shorthand form which is modelled by other SDL constructs (see *Model* below).

Concrete graphical grammar

The concrete graphical syntax is specified in the extended Backus-Naur Form of syntax description defined in § 1.5.3.

The graphical syntax is followed by paragraphs defining the static conditions which must be satisfied in well-formed SDL/GR and which can be checked without interpretation of an instance. Static conditions (if any) for the abstract grammar also apply.

The relationship between concrete graphical syntax and abstract syntax is omitted if the topic being defined is a shorthand form which is modelled by other SDL constructs (see *Model* below).

In many cases there is a simple relationship between concrete graphical grammar diagrams and abstract syntax definitions. When the name of a non-terminal ends in the concrete grammar with the word "diagram" and there is a name in the abstract grammar which differs only by ending in the word *definition*, then the two rules represent the same concept. For example, <system diagram> in the concrete grammar and *System-definition* in the abstract grammar correspond.

Expansion in the concrete syntax arising from such facilities as remote definitions (§ 2.4.1), macros (§ 4.2) and literals mappings (§ 5.4.1.15) etc., must be considered before the correspondence between the concrete and the abstract syntax.

Semantics

Properties are used in the well-formedness rules which involve either the type or other types which refer to that type.

An example of a property is the set of valid input signal identifiers of a process. This property is used in the static condition "For each *state-node*, all input *signal-identifiers* (in the valid input signal set) appear in either a *Save-signalset* or an *Input-node*."

All instances have an identity property but unless this is formed in some unusual way this identity property is determined as defined by the general section on identities in § 2. Therefore this is not usually mentioned as an identity property. Also it has not been necessary to mention sub-components of definitions contained by the definition since the ownership of such sub-components is obvious from the abstract syntax. For example it is obvious that a block definition "has" enclosed process definitions and/or a block substructure definition.

Properties are static if they can be determined without interpretation of an SDL system specification and are dynamic if an interpretation of the same is required to determine the property.

The interpretation is described in an operational manner. Whenever there is a list in the Abstract Syntax, the list is interpreted in the order given. That is, the Recommendation describes how the instances are created from the system definition and how these instances are interpreted within an "abstract SDL machine".

Dynamic conditions are conditions which must be satisfied during interpretation and cannot be checked without interpretation. Dynamic conditions may lead to errors (see § 1.3.3).

Model

Some constructs are considered to be "derived concrete syntax" (or a shorthand) for other equivalent concrete syntax constructs. For example omitting an input for a signal is derived concrete syntax for an input for that signal followed by a null transition back to the same state.

Sometimes such "derived concrete syntax", if expanded, would give rise to an extremely large (possibly infinite) representation. Nevertheless, the semantics of such a specification can be determined.

Examples

The titled enumeration item *Examples* contains examples.

1.5 *Metalanguages*

For the definition of properties and syntaxes of SDL different metalanguages have been used according to the particular needs.

In the following an introduction of the metalanguages used is given; where appropriate only references to textbooks or specific ITU publications are given.

1.5.1 *Meta IV*

The following subset of Meta IV is used to describe the abstract syntax of SDL.

A definition in the abstract syntax can be regarded as a named composite object (a tree) defining a set of sub-components.

For example the abstract syntax for variable definition is

Variable-definition :: *Variable-name Sort-reference-identifier*

which defines the domain for the composite object (tree) named *Variable-definition*. This object consists of two sub-components which in turn might be trees.

The Meta IV definition

Sort-reference-identifier = *Identifier*

expresses that a *Sort-reference-identifier* is an *Identifier* and cannot therefore syntactically be distinguished from other identifiers.

An object might also be of some elementary (non-composite) domains. In the context of SDL these are:

a) Integer objects

example

Number-of-instances :: *Intg Intg*

Number-of-instances denotes a composite domain containing two integer (*Intg*) values denoting the initial number and the maximum number of instances.

b) Quotation objects

These are represented as any bold face sequence of uppercase letters and digits.

example

Destination-process = *Process-identifier* | **ENVIRONMENT**

The *Destination-process* is either a *Process-identifier* or the environment which is denoted by the quotation **ENVIRONMENT**.

c) Token objects

Token denotes the domain of tokens. This domain can be considered as consisting of a potentially infinite set of distinct atomic objects for which no representation is required.

example

Name :: *Token*

A name consists of an atomic object such that any *Name* can be distinguished from any other name.

d) Unspecified objects

An unspecified object denotes domains which might have some representation, but for which the representation is of no concern in this Recommendation.

example

Informal-text :: ...

Informal-text contains an object which is not interpreted.

The following operators (constructors) in BNF (see §1.5.2) are also used in the abstract syntax: "*" for possible empty list, "+" for non-empty list, "|" for alternative, and "[" "]" for optional.

Parentheses are used for grouping of domains which are logically related.

Finally, the abstract syntax uses another postfix operator "-**set**" yielding a set (unordered collection of distinct objects) . Example

Process-graph :: *Process-start-node State-node*-**set**

A *Process-graph* consists of a *Process-start-node* and a set of *State-nodes*

1.5.2 *BNF*

In the Backus Naur Form a terminal symbol is either indicated by not enclosing it within angular brackets (that is the less-than sign and greater-than sign, < and >) or it is one of the two representations <name> and <character string>. Note that the two special terminals <name> and <character string> may also have semantics stressed as defined below.

The angular brackets and enclosed word(s) are either a non-terminal symbol or one of the two terminals <character string> or <name>. Syntactic categories are the non-terminals indicated by one or more words enclosed between angular brackets. For each non-terminal symbol, a production rule is given either in concrete textual grammar or in graphical grammar. For example

<view expression> ::=
VIEW (<variable identifier>, <expression>)

A production rule for a non-terminal symbol consists of the non-terminal symbol at the

left-hand side of the symbol ::=, and one or more constructs, consisting of non-terminal and/or terminal symbol(s) at the right-hand side. E.g. <view expression>, <variable identifier> and <expression> in the example above are non-terminals; VIEW, the parentheses and the comma are terminal symbols.

Sometimes the symbol includes an underlined part. This underlined part stresses a semantic aspect of that symbol. E.g. <variable identifier> is syntactically identical to <identifier>, but semantically it requires the identifier to be a variable identifier.

At the right-hand side of the ::= symbol several alternative productions for the non-terminal can be given, separated by vertical bars (|). For example

```
<block area> ::=
                    <graphical block reference>
               |    <block diagram>
```

expresses that a <block area> is either a <graphical block reference> or a <block diagram>.

Syntactic elements may be grouped together by using curly brackets ({ and }), similar to the parentheses in Meta IV (see § 1.5.1). A curly bracketed group may contain one or more vertical bars, indicating alternative syntactic elements. For example

```
<block interaction area> ::=
                    {<block area> | <channel definition area>}+
```

Repetition of curly bracketed groups is indicated by an asterisk (*) or plus sign (+). An asterisk indicates that the group is optional and can be further repeated any number of times; a plus sign indicates that the group must be present and can be further repeated any number of times. The example above expresses that a <block interaction area> contains at least one <block area> or <channel definition area> and may contain several more <block area>s and <channel definition area>s.

If syntactic elements are grouped using square brackets ([and]), then the group is optional. For example

```
<process heading> ::=
                    PROCESS <process identifier> [<formal parameters>]
```

expresses that a <process heading> may, but need not, contain <formal parameters>.

1.5.3 *Metalanguage for graphical grammar*

For the graphical grammar the metalanguage described in § 1.5.2 is extended with the following metasymbols:

a) **contains**
b) **is associated with**
c) **is followed by**
d) **is connected to**
e) **set**

The **set** metasymbol is a postfix operator operating on the immediately preceding syntactic elements within curly brackets, and indicating an (unordered) set of items. Each item may be any

group of syntactic elements, in which case it must be expanded before applying the **set** metasymbol.

Example:

{{<system text area>}* {<macro diagram>}* <block interaction area>} **set**

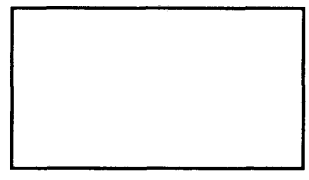
is a set of zero or more <system text area>s, zero or more <macro diagram>s and one <block interaction area>.

All the other metasymbols are infix operators, having a graphical non-terminal symbol as the left-hand argument. The right-hand argument is either a group of syntactic elements within curly brackets or a single syntactic element. If the right-hand side of a production rule has a graphical non-terminal symbol as the first element and contains one or more of these infix operators, then the graphical non-terminal symbol is the left-hand argument of each of these infix operators. A graphical non-terminal symbol is a non-terminal having the word "symbol" immediately before the greater than sign >.

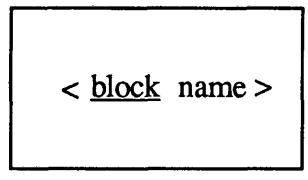
The metasymbol **contains** indicates that its right-hand argument should be placed within its left-hand argument and the attached <text extension symbol>, if any. Example:

<graphical block reference> ::=
<block symbol> **contains** <block name>

<block symbol> ::=



means the following



The metasymbol **is associated with** indicates that its right-hand argument is logically associated with its left-hand argument (as if it were "contained" in that argument, the unambiguous association is ensured by appropriate drawing rules).

The metasymbol **is followed by** means that its right-hand argument follows (both logically and in drawing) its left-hand argument.

The metasymbol **is connected to** means that its right-hand argument is connected (both logically and in drawing) to its left-hand argument.

2 Basic SDL

2.1 *Introduction*

An SDL system has a set of blocks. Blocks are connected to each other and to the environment by channels. Within each block there are one or more processes. These processes communicate with one another by signals and are assumed to execute concurrently.

§ 2 has been been divided into eight main topics:

a) *General rules*

basic SDL concepts such as lexical rules and identifiers, visibility rules, informal text, partitioning of diagrams, drawing rules, comments, text extensions, text symbols.

b) *Basic data concepts*

basic SDL data concepts such as values, variables, expressions.

c) *System structure*

contains SDL concepts dealing with the general structuring concepts of the language. Such concepts are system, block, process, procedure.

d) *Communication*

contains communication mechanisms used in SDL such as channel, signal route, signal.

e) *Behaviour*

the constructs that are relevant to the behaviour of a process: general connectivity rules of a process or procedure graph, variable definition, start, state, input, save, label, transition.

f) *Action*

active constructs such as task, process create, procedure call, output, decision.

g) *Timers*

Timer definition and Timer primitives.

h) *Examples*

examples referred to from the other topics.

2.2 *General rules*

2.2.1 *Lexical rules*

Lexical rules define lexical units. Lexical units are the terminal symbols of the *Concrete textual syntax.*

<lexical unit> ::=
<name>
| <character string>
| <special>
| <composite special>
| <note>
| <keyword>

<name> ::=
<word> {<underline> <word> }*

<word> ::=
{<alphanumeric> | <full stop>}*
<alphanumeric>
{<alphanumeric> | <full stop>}*

<alphanumeric> ::=
<letter>
| <decimal digit>
| <national>

<letter> ::=
A | B | C | D | E | F | G | H | I | J | K | L | M
| N | O | P | Q | R | S | T | U | V | W | X | Y | Z
| a | b | c | d | e | f | g | h | i | j | k | l | m
| n | o | p | q | r | s | t | u | v | w | x | y | z

<decimal digit> ::=
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<national> ::=
#
| `
| ¤
| @
| <left square bracket>
| \
| <right square bracket>
| <left curly bracket>
| <vertical line>
| <right curly bracket>
| <overline>
| <upward arrow head>

```
<left square bracket> ::=
            [

<right square bracket> ::=
            ]

<left curly bracket> ::=
            {

<vertical line> ::=
            |

<right curly bracket> ::=
            }

<overline>  ::=
            ~

<upward arrow head>  ::=
            ^

<full stop> ::=
            .

<underline> ::=
            _

<character string> ::=
                  <apostrophe> {<alphanumeric>
                              |     <other character>
                              |     <special>
                              |     <full stop>
                              |     <underline>
                              |     <space>
                              |     <apostrophe> <apostrophe> }* <apostrophe>

<text> ::=
                  {<alphanumeric>
            |     <other character>
            |     <special>
            |     <full stop>
            |     <underline>
            |     <space>
            |     <apostrophe>}*


<apostrophe> ::=
            '

<other character> ::=
            ? | &|%

<special> ::=
            + | - | ! | / | > | * | ( | ) | " | , | ; | < | = | :
```

<composite special> ::=
        ==
    | ==>
    | /=
    | <=
    | >=
    | //
    | :=
    | =>
    | ->
    | (.
    | .)

<note> ::=
        /* <text> */

<keyword> ::=
        ACTIVE
    | ADDING
    | ALL
    | ALTERNATIVE
    | AND
    | AXIOMS
    | BLOCK
    | CALL
    | CHANNEL
    | COMMENT
    | CONNECT
    | CONSTANT
    | CONSTANTS
    | CREATE
    | DCL
    | DECISION
    | DEFAULT
    | ELSE
    | ENDALTERNATIVE
    | ENDBLOCK
    | ENDCHANNEL
    | ENDDECISION
    | ENDGENERATOR
    | ENDMACRO
    | ENDNEWTYPE
    | ENDPROCEDURE
    | ENDPROCESS
    | ENDREFINEMENT
    | ENDSELECT
    | ENDSERVICE
    | ENDSTATE
    | ENDSUBSTRUCTURE
    | ENDSYNTYPE
    | ENDSYSTEM
    | ENV
    | ERROR
    | EXPORT
    | EXPORTED

EXTERNAL
FI
FOR
FPAR
FROM
GENERATOR
IF
IMPORT
IMPORTED
IN
INHERITS
INPUT
JOIN
LITERAL
LITERALS
MACRO
MACRODEFINITION
MACROID
MAP
MOD
NAMECLASS
NEWTYPE
NEXTSTATE
NOT
NOW
OFFSPRING
OPERATOR
OPERATORS
OR
ORDERING
OUT
OUTPUT
PARENT
PRIORITY
PROCEDURE
PROCESS
PROVIDED
REFERENCED
REFINEMENT
REM
RESET
RETURN
REVEALED
REVERSE
SAVE
SELECT
SELF
SENDER
SERVICE
SET
SIGNAL
SIGNALLIST
SIGNALROUTE
SIGNALSET
SPELLING

| START
| STATE
| STOP
| STRUCT
| SUBSTRUCTURE
| SYNONYM
| SYNTYPE
| SYSTEM
| TASK
| THEN
| TIMER
| TO
| TYPE
| VIA
| VIEW
| VIEWED
| WITH
| XOR

The <space> represents the CCITT Alphabet No 5 character for a space.

The <national> characters are represented above as in the International Reference Version of CCITT Alphabet No. 5 (Recommendation T.50). The responsibility for defining the national representations of these characters lies with national standardisation bodies.

All <letter>s are always treated as if uppercase, except within <character string>. (The treatment of <national>s may be defined by national standardisation bodies.)

A <lexical unit> is terminated by the first character which cannot be part of the <lexical unit> according to the syntax specified above. When an <underline> character is followed by one or more control characters (control characters are defined as in Recommendation T.50) or spaces, all of these characters (including the <underline>) are ignored, e.g. A_ B denotes the same <name> as AB. This use of <underline> allows <lexical unit>s to be splitted over more than one line.

When an <underline> character is followed by a <word> in a <name>, it is allowed to specify one or more control characters or spaces instead of the <underline> character, as long as one of the <word>s enclosing the <underline> character does not form a <keyword>, e.g. A B denotes the same <name> as A_B.

However, there are some cases where the absence of <underline> in <name>s is ambiguous. The following rules therefore apply:

1. The <underline>s in the <name> in a <path item> must be specified explicitly.

2. When one or more <name>s or <identifier>s may be followed directly by a <sort> (e.g. <variable definition>s, <view definition>s) then the <underline>s in these <name>s or <identifier>s must be specified explicitly.

3. When a <data definition> contains <generator instantiations> then the <underline>s in the <sort name> following the keyword NEWTYPE must be specified explicitly.

A control character has the same meaning of a space.

Control characters and spaces may appear any number of times between two <lexical unit>s. Any number of control characters and spaces between two <lexical unit>s has the same meaning as one space.

The character / immediately followed by the character * always starts a <note>.The character * immediately followed by the character / in a <note> always terminates the <note>. A <note> may be inserted before or after any <lexical unit>.

Special lexical rules apply within a <macro body> (see § 4.2.1).

2.2.2 *Visibility rules and identifiers*

Abstract grammar

Identifier	::	*Qualifier Name*
Qualifier	=	*Path-item* +
Path-item	=	*System-qualifier* |
		Block-qualifier |
		Block-substructure-qualifier |
		Signal-qualifier |
		Process-qualifier |
		Procedure-qualifier |
		Sort-qualifier
System-qualifier	::	*System-name*
Block-qualifier	::	*Block-name*
Block-substructure-qualifier	::	*Block-substructure-name*
Process-qualifier	::	*Process-name*
Procedure-qualifier	::	*Procedure-name*
Signal-qualifier	::	*Signal-name*
Sort-qualifier	::	*Sort-name*
Name	::	*Token*

Concrete textual grammar

<identifier> ::=
[<qualifier>] <name>

<qualifier> ::=
<path item> {/<path item>}*

<path item> ::=
<scope unit class> <name>

<scope unit class> ::=
SYSTEM
| BLOCK
| SUBSTRUCTURE
| SIGNAL
| PROCESS
| PROCEDURE

| TYPE
| SERVICE

There is no corresponding abstract syntax for the <scope unit class> denoted by SERVICE. The <name>s and <identifier>s of entities defined in a <service definition> are transformed into unique <name>s respectively <identifier>s defined in the <process definition> containing the <service definition>.

The <qualifier> reflects the hierarchical structure from the system level to the defining context, and such that the system level is the textual leftmost part.

It is allowed to omit some of the leftmost <path item>s (except for <remote definition>s, see § 2.4.1), or the whole <qualifier>. When the whole <qualifier> is omitted and the <name> denotes an entity of the entity class containing variables, synonyms, literals and operators (see *Semantics* below), the binding of the <name> to a definition must be resolvable by the actual context. In other cases the <identifier> is bound to an entity that has its defining context in the nearest enclosing scope unit in which the <qualifier> of the <identifier> is the same as the rightmost part of the full <qualifier> denoting this scope unit. If the <identifier> does not contain a <qualifier>, then the requirement on matching of <qualifier>s is omitted.

A subsignal must be qualified by its parent signal unless no other visible signal exists at that place which have the same <name>.

Resolution by context is possible in the following cases:

a) The scope unit in which the <name> is used is not a <partial type definition> and it contains a definition having that <name>. The <name> will be bound to that definition.

b) The scope unit in which the <name> is used does not contain any definition having that <name> or the scope unit is a <partial type definition>, and in the whole <system definition> there exists exactly one visible definition of an entity that has the same <name> and to which the <name> can be bound without violating any static properties (sort compatibilty etc) of the construct in which the <name> occurs. The <name> will be bound to that definition.

Only visible identifiers may be used, except for the <variable identifier> in a <view definition> and for the <identifier> used in place of a <name> in a referenced definition (that is a definition taken out from the <system definition>).

Semantics

Scope units are defined by the following schema:

Concrete textual grammar	*Concrete graphical grammar*
<system definition>	<system diagram>
<block definition>	<block diagram>
<process definition>	<process diagram>
<procedure definition>	<procedure diagram>
<block substructure definition>	<block substructure diagram>

<channel substructure definition> <channel substructure diagram>

<service definition> <service diagram>

<partial type definition>

<signal refinement>

A scope unit has a list of definitions attached. Each of the definitions defines an entity belonging to a certain entity class and having an associated name. For a <partial type definition>, the attached list of definitions consists of the <operator signature>s, the <literal signature>s and any <operator signature> and <literal signature>s inherited from a parent sort, from a generator instance or implied by the use of shorthand notations such as the keyword ORDERING (see § 5.4.1.8). Note, that a <view definition> does not define an entity.

Although <infix operator>s, <operator>s with an exclamation and <character string>s have their own syntactical notation they are in fact <name>s, they are in the *Abstract syntax* represented by a *name*. In the following, they are treated as if they (also syntactically) were <name>s. However <state name>s, <connector name>s, <generator formal name>s, <value identifier>s in equations, <macro formal name>s and <macro name>s have special visibility rules and cannot therefore be qualified. <state name>s and <connector name>s are not visible outside a <process body>, <procedure body> or <service body> respectively. Other special visibility rules are explained in the appropriate sections.

Each entity is said to have its defining context in the scope unit which defines it. Entities are referenced by means of <identifier>s.

The <qualifier> within an <identifier> specifies uniquely the defining context of the <name>.

The following entity classes exist:

a) system
b) blocks
c) channels, signal routes
d) signals, timers
e) processes
f) procedures
g) variables (including formal parameters), synonyms, literals, operators
h) sorts
i) generators
j) imported entities
k) signal lists
l) services
m) block substructures, channel substructures

An <identifier> is said to be visible in a scope unit

a) if the name part of the <identifier> has its defining context in that scope unit, or
b) if it is visible in the scope unit which defined that scope unit, or
c) if the scope unit contains a <partial type definition> in which the <identifier> is defined, or
d) if the scope unit contains a <signal definition> in which the <identifier> is defined.

No two definitions in the same scope unit and belonging to the same entity class can have the same <name>. An exception is <operator signature> and <literal signature> definitions in the same

<partial type definition> (see § 5.2.2): two or more operators and/or literals can have the same <name> with different <arguments sort>s or different <result> sort.
Another exception is imported entities. For this entity class the pairs of (<import name>,<sort>) in <import definition>s in the scope unit must be distinct.

In the concrete textual grammar, the optional name or identifier in a definition after the ending keywords (ENDSYSTEM, ENDBLOCK, etc.) must be syntactically the same as the name or identifier following the corresponding commencing keyword (SYSTEM, BLOCK, etc. respectively).

2.2.3 *Informal text*

Abstract grammar

Informal-text :: ...

Concrete textual grammar

<informal text> ::=
<character string>

Semantics

If informal text is used in an SDL system specification, it means that this text is not formal SDL, i.e., SDL does not give it any semantics. The semantics of the informal text can be defined by some other means.

2.2.4 *Drawing rules*

The size of the graphical symbols can be chosen by the user.

Symbol boundaries must not overlay or cross. An exception to this rule applies for line symbols, i.e. <channel symbol>, <signal route symbol>, <create line symbol>, <flow line symbol>, <solid association symbol> and <dashed association symbol>, which may cross each other. There is no logical association between symbols which do cross.

The metasymbol **is followed by** implies a <flow line symbol>.

Line symbols may consist of one or more straight line segments.

An arrowhead is required on a <flow line symbol>, when it enters another <flow line symbol>, an <out-connector symbol> or a <nextstate symbol>. In other cases, arrowheads are optional on <flow line symbol>s. The <flow line symbol>s are horizontal or vertical.

Vertical mirror images of <input symbol>, <output symbol>, <comment symbol> and <text extension symbol> are allowed.

The righthand argument of the metasymbol **is associated with** must be closer to the lefthand argument than to any other graphical symbol. The syntactical elements of the righthand argument must be distinguishable from each other.

Text within a graphical symbol must be read from left to right, starting from the upper left corner. The righthand edge of the symbol is interpreted as a newline character, indicating that the reading must continue at the leftmost point of the next line (if any).

2.2.5 *Partitioning of diagrams*

The following definition of diagram partitioning is not part of the *Concrete graphical grammar*, but the same metalanguage is used.

<page>::=
    <frame symbol> **contains**
    {<heading area> <page number area>
    {<syntactical unit>}*}

<heading area> ::=
        <implicit text symbol> **contains** <heading>

<page number area> ::=
        <implicit text symbol> **contains** [<page number> [(<number of pages>)]]

<page number> ::=
        <literal name>

<number of pages> ::=
            <natural literal name>

The <page> is a starting non-terminal, therefore it is not referred to in any production rule. A diagram may be partitioned into a number of <page>s, in which case the <frame symbol> delimiting the diagram and the diagram <heading> are replaced by a <frame symbol> and a <heading> for each <page>.

The user of SDL may choose <frame symbol>s to be implied by the boundary of the media on which diagrams are reproduced.

The <implicit text symbol> is not shown, but implied, in order to have a clear separation between <heading area> and <page number area>. The <heading area> is placed at the upper left corner of the <frame symbol>. <page number area> is placed at the upper right corner of the <frame symbol>. <heading> and <syntactical unit> depends on the type of diagram.

2.2.6 *Comment*

A comment is a notation to represent comments associated with symbols or text.

In the *Concrete textual grammar* two forms of comments are used. The first form is the <note> defined in § 2.2.1.

Examples are shown in Figure 2.9.1 and in Figure 2.9.3.

The concrete syntax of the second form is:

<end> ::=
        [<comment>];

<comment> ::=
        COMMENT <character string>

An example is shown in Figure 2.9.2.

In the *Concrete graphical grammar* the following syntax is used:

<comment area> ::=
<comment symbol> **contains** <text>
is connected to <dashed association symbol>

<comment symbol> ::=

<dashed association symbol> ::=

One end of the <dashed association symbol> must be connected to the middle of the vertical segment of the <comment symbol>.

A <comment symbol> can be connected to any graphical symbol by means of a <dashed association symbol>. The <comment symbol> is considered as a closed symbol by completing (in imagination) the rectangle to enclose the text. It contains comment text related to the graphical symbol.

An example is shown in Figure 2.9.4 in § 2.9.

2.2.7 *Text extension*

<text extension area> ::=
<text extension symbol> **contains** <text>
is connected to <solid association symbol>

<text extension symbol> ::=
<comment symbol>

<solid association symbol> ::=

One end of the <solid association symbol> must be connected to the middle of the vertical segment of the <text extension symbol>.

A <text extension symbol> can be connected to any graphical symbol by means of a <solid association symbol>. The <text extension symbol> is considered as a closed symbol by completing (in imagination) the rectangle.

The text contained in the <text extension symbol> is a continuation of the text within the graphical symbol and is considered to be contained in that symbol.

2.2.8 *Text symbol*

<text symbol> is used in any <diagram>. The content depends on the diagram.

<text symbol> ::=

2.3 *Basic data concepts*

The concept of data in SDL is defined in §5; that is the SDL data terminology, the facility to define new data types and predefined data facilities.

Occurrences of data are in data type definitions, expressions, the application of operators, variables, values and literals.

2.3.1 *Data type definitions*

Data in SDL is principally concerned with data types. A data type defines sets of values, a set of operators which can be applied to these values, and a set of algebraic rules (equations) defining the behaviour of these operators when applied to the values. The values, operators and algebraic rules collectively define the properties of the data type. These properties are defined by data type definitions.

SDL allows the definition of any needed data type, including composition mechanisms (composite types), subject only to the requirement that such a definition can be formally specified. By contrast, for programming languages there are implementation considerations which require that the set of available data types and, in particular, the composition mechanisms provided (array, structure, etc.) be limited.

2.3.2 *Variable*

Variables are objects which can be associated with a value by assignment. When the variable is accessed, the associated value is returned.

2.3.3 *Values and literals.*

A set of values with certain characteristics is called a sort. Operators are defined from and to values of sorts. For instance the application of the operator for summation ("+") from and to values of the Integer sort is valid, whereas summation of the Boolean sort is not.

All sorts have at least one value. Each value belongs to one and only one sort, that is sorts never have values in common.

For most sorts there are literal forms to denote values of the sort (for example for Integers "2" is used rather than "1 + 1". There may be more than one literal to denote the same value (for example 12 and 012 can be used to denote the same Integer value). The same literal denotation may be used for more than one sort; for example 'A' is both a character and a character string of length one. Some sorts may have no literals; for example, a composite value often has no literals of its own but has its values defined by composition operations on values of its components.

2.3.4 *Expressions*

An expression denotes a value. If an expression does not contain a variable, for instance if it is a literal of a given sort, each occurrence of the expression will always denote the same value. An expression which contains variables may be interpreted as different values during the interpretation

of an SDL system depending on the value associated with the variables.

2.4 *System structure*

2.4.1 *Remote definitions*

A <remote definition> is a definition that has been removed from its defining context to gain overview. It is similar to a macro definition (see § 4.2), but it is "called" from exactly one place (the defining context) using a reference.

Concrete grammar

<remote definition> ::=
<definition> | <diagram>

< system definition> ::=
{<textual system definition> | <system diagram>}
{<remote definition>}*

<definition> ::=

<block definition>
| <process definition>
| <procedure definition>
| <block substructure definition>
| <channel substructure definition>
| <service definition>
| <macro definition>

<diagram> ::=
<block diagram>
| <process diagram>
| <procedure diagram>
| <block substructure diagram>
| <channel substructure diagram>
| <service diagram>
| <macro diagram>

For each <remote definition>, except for <macro definition> and <macro diagram> there must be a reference in the <system definition>, the <system diagram>, or another <remote definition>.

For each reference there must be a corresponding <remote definition>.

In each <remote definition> there must be an <identifier> immediately after the initial keyword. The <qualifier> in this <identifier> must be either complete, or omitted. If the <qualifier> is omitted, the <name> must be unique in the system definition, within the entity class for the <remote definition>. It is not allowed to specify a <qualifier> in the <identifier> after the initial keyword for definitions which are not <remote definition>s (i.e. a <name> must be specified for normal definitions).

Semantics

Before a <concrete system definition> can be analyzed, each reference must be replaced by the corresponding <remote definition>. In this replacement the <identifier> of the <remote definition> is replaced by the <name> in the reference.

2.4.2 *System*

Abstract grammar

System-definition	::	*System-name* *Block-definition*-**set** *Channel-definition*-**set** *Signal-definition*-**set** *Data-type-definition* *Syn-type-definition*-**set**
System-name	=	*Name*

A *System-definition* has a name which can be used in qualifiers.

There must be at least one *Block-definition* contained in the *System-definition.*

The definitions of all the signals, channels, data types, syntypes, used in the interface with the environment and between blocks of the system are contained in the *System-definition.*
All predefined data are regarded to be defined at system level.

Concrete textual grammar

```
<textual system definition> ::=
                        SYSTEM <system name> <end>
                        {<block definition>
                        |    <textual block reference>
                        |    <channel definition>
                        |    <signal definition>
                        |    <signal list definition>
                        |    <select definition>
                        |    <macro definition>
                        |    <data definition>}+
                                ENDSYSTEM [<system name>] <end>

<textual block reference> ::=
                        BLOCK <block name> REFERENCED <end>
```

The <select definition> is defined in § 4.3.3, <macro definition> in § 4.2, <data definition> is defined in § 5.5.1, <block definition> is defined in § 2.4.3 <channel definition> is defined in § 2.5.1. <signal definition> is defined in § 2.5.4. <signal list definition> is defined in § 2.5.5.

An example of <system definition> is shown in Figure 2.9.5 in § 2.9.

Concrete graphical grammar

<system diagram> ::=
<frame symbol> **contains**
{<system heading>
{ {<system text area>}*
{<macro diagram>}*
<block interaction area> }**set** }

<frame symbol> ::=

<system heading> ::=
SYSTEM <system name>

<system text area> ::=
<text symbol> **contains**
{ <signal definition>
| <signal list definition>
| <data definition>
| <macro definition>
| <select definition>}*

<block interaction area> ::=
{<block area>
| <channel definition area>}+

<block area> ::=
<graphical block reference>
| <block diagram>

<graphical block reference> ::=
<block symbol> **contains** <block name>

<block symbol> ::=

The <select definition> is defined in § 4.3.3, <macro definition> and <macro diagram> in § 4.2, <data definition> is defined in § 5.5.1, <block diagram> is defined in § 2.4.3 <channel definition area> is defined in § 2.5.1. <signal definition> is defined in § 2.5.4. <signal list definition> is defined in § 2.5.5.

The *Block-definition*-**set** in the *Abstract grammar* corresponds to the <block area>s, the *Channel-definition*-**set** corresponds to the <channel definition area>.

An example of a <system diagram> is shown in Figure 2.9.6.

Semantics

A *System-definition* is the SDL representation of a specification or description of a system.

A system is separated from its environment by a system boundary and contains a set of blocks. Communication between the system and the environment or between blocks within the system can only take place using signals. Within a system, these signals are conveyed on channels. The channels connect blocks to one another or to the system boundary.

Before interpreting a *System-definition* a consistent subset (see § 3.2.1) is chosen. This subset is called an instance of the *System-definition.* A system instance is an instantiation of a system type defined by a *System-definition.*The interpretation of an instance of a *System-definition* is performed by an abstract SDL machine which thereby gives semantics to the SDL concepts. To interpret an instance of a *System-definition* is to:

a) to initiate the system time
b) to interpret the blocks and their connected channels which are contained in the consistent partitioning subset selected.

2.4.3 *Block*

Abstract grammar

Block-definition	::	*Block-name* *Process-definition*-**set** *Signal-definition*-**set** *Channel-to-route-connection*-**set** *Signal-route-definition*-**set** *Data-type-definition* *Syn-type-definition*-**set** [*Block-substructure-definition*]
Block-name	=	*Name*

Unless a *Block-definition* contains a *Block-substructure-definition* there must be at least one *Process-definition* and *Signal-route-definition* within the block.

It is possible to perform partitioning activities on the blocks specifying *Block-substructure-definition*; this feature of the language is treated in § 3.2.2

Concrete textual grammar

<block definition> ::=
BLOCK {<block name>|<block identifier>} <end>
{<signal definition>
| <signal list definition>
| <process definition>
| <textual process reference>
| <signal route definition>
| <macro definition>
| <data definition>
| <select definition>
| <channel to route connection>}*
[<block substrucure definition>|<textual block substructure reference>]
ENDBLOCK [<block name>| <block identifier>] <end>

<textual process reference> ::=
PROCESS <process name> [<number of instances>] REFERENCED <end>

<signal definition> is defined in § 2.5.4, <signal list definition> in § 2.5.5, <process definition> in § 2.4.4, <signal route definition> in § 2.5.2, <channel to route connection> in § 2.5.3. <block substructure definition> and <textual block substructure reference> are defined in § 3.2.2, <macro definition> in § 4.2.2 and <data definition> in § 5.5.1.

An example of <block definition> is shown in Figure 2.9.7 in § 2.9.

Concrete graphical grammar

<block diagram> ::=
<frame symbol>
contains {<block heading>
{ {<block text area>}* {<macro diagram>}*
[<process interaction area>] [<block substructure area>]}**set** }
is associated with {<channel identifier>}*

The <channel identifier> identifies a channel connected to a signal route in the <block diagram>. It is placed outside the <frame symbol> close to the endpoint of the signal route at the <frame symbol>. If the <block diagram> does not contain a <process interaction area>, then it must contain a <block substructure area>.

<block heading> ::=
BLOCK {<block name> | <block identifier> }

<block text area> ::=
<system text area>

<process interaction area> ::=
{ <process area>
| <create line area>
| <signal route definition area>}+

<process area> ::=
<graphical process reference> | <process diagram>

<graphical process reference> ::=
<process symbol> **contains** { <process name> [<number of instances>]}

<process symbol> ::=

<number of instances> is defined in § 2.4.4.

<create line area> ::=
<create line symbol>
is connected to {<process area> <process area>}

<create line symbol> ::=

The arrowhead on the <create line symbol> indicates the <process area> upon which the create action is performed.

The <process diagram> is defined in § 2.4.4, <signal route definition area> in § 2.5.2, <block substructure area> in § 3.2.2, <macro diagram> in § 4.2.2.

An example of <block diagram> is shown in Figure 2.9.8 in § 2.9.

Semantics

A block definition is a container for one or more process definitions of a system and/or a block substructure. Purpose of the block definition is the grouping of processes that as a whole perform a certain function, either directly or by a block substructure.

A block definition provides a static communication interface by which its processes can communicate with other processes.In addition it provides the scope for process definitions.

To interpret a block is to create the initial processes in the block.

2.4.4 *Process*

Abstract grammar

Process-definition	::	*Process-name* *Number-of-instances* *Process-formal-parameter* ***** *Procedure-definition*-**set** *Signal-definition*-**set** *Data-type-definition* *Syn-type-definition*-**set** *Variable-definition*-**set** *View-definition*-**set** *Timer-definition*-**set** *Process-graph*
Number-of-instances	::	*Intg Intg*
Process-name	=	*Name*
Process-graph	::	*Process-start-node* *State-node*-**set**
Process-formal-parameter	::	*Variable-name* *Sort-reference-identifier*

Concrete textual grammar

```
<process definition> ::=
                        PROCESS {<process identifier>| <process name> }
                            [<number of instances>] <end>
                                [<formal parameters> <end>] [<valid input signal set>]
                                  {<signal definition>
                                |  <signal list definition>
                                |  <procedure definition>
                                |  <textual procedure reference>
                                |  <macro definition>
                                |  <data definition>
                                |  <variable definition>
                                |  <view definition>
                                |  <select definition>
                                |  <import definition>
                                |  <timer definition>} *
                                       {<process body>
                                       |  <service decomposition>}
                        ENDPROCESS [ <process name>| <process identifier> ] <end>

<textual procedure reference>  ::=
                        PROCEDURE <procedure name> REFERENCED <end>

<valid input signal set>  ::=
                        SIGNALSET [<signal list>] <end>

<process body>  ::=
                    <start> {<state>} *

<formal parameters>  ::=
                        FPAR <variable name> {, <variable name>}*<sort>
                                {,<variable name>{, <variable name>}*<sort>}*

<number of instances>  ::=
                        ([<initial number>],[<maximum number>])

<initial number>  ::=
                    <natural simple expression>

<maximum number>  ::=
                        <natural simple expression>
```

The initial number of instances and maximum number of instances contained in *Number-of-instances* are derived from <number of instances>. If <initial number> is left out then <initial number> is 1. If <maximum number> is omitted then <maximum number> is unbounded.

The <number of instances> used in the derivation is the following:

a) If there is no <textual process reference> for the process then the <number of instances> in the <process definition> is used. If it does not contain a <number of instances> then the <number of instances> where both < initial number> and <maximum number> are omitted is used.

b) If both the <number of instances> in <process definition> and the <number of instances> in a <textual process reference> are omitted then the <number of instances> where both <initial number> and <maximum number> are omitted is used.

c) If either the <number of instances> in <process definition> or the <number of instances> in a <textual process reference> are omitted then the <number of instances> is the one which is present.

d) If both the <number of instances> in <process definition> and the <number of instances> in a <textual process reference> are specified then the two <number of instances> must be equal lexically and this <number of instances> is used.

Similar relation applies for <number of instances> specified in <process diagram> and in <graphical process reference> as defined below.

The <signal definition > is defined in § 2.5.4, <signal list definition> in § 2.5.5, <view definition> in § 2.6.1.2, <variable definition> in § 2.6.1.1, <procedure definition> in § 2.4.5, <timer definition> in § 2.8, <macro definition> in § 4.2.2, <import definition> in § 4.1.3, <select definition> in § 4.3.3, <simple expression> in § 4.3.2 <service decomposition> in § 4.10.1, <data definition> in § 5.5.1.

The <initial number> of instances must be less than or equal to <maximum number> and <maximum number> must be greater than zero.

The use of <valid input signal set> is defined in § 2.5.2 *Model.*

An example of <process definition> is shown in Figure 2.9.9 in § 2.9.

Concrete graphical grammar

<process diagram> ::=
<frame symbol>
contains {<process heading>
{ {<process text area>}*
{<procedure area>}*
{<macro diagram> }*
{<process graph area> | <service interaction area> } }**set** }
[is associated with {<signal route identifier>}+]

The <signal route identifier> identifies an external signal route connected to a signal route in the <process diagram>. It is placed outside the <frame symbol> close to the endpoint of the signal route at the <frame symbol>.

<process text area> ::=
<text symbol> **contains** {
[<valid input signalset>]
{<signal definition>
| <signal list definition>
| <variable definition>
| <view definition>
| <import definition>
| <data definition>
| <macro definition>
| <timer definition>
| <select definition> }* }

<process heading> ::=
PROCESS {<process name>|<process identifier>}
[<number of instances> [<end>]]
[<formal parameters>]

<process graph area> ::=
<start area> { <state area> |<in-connector area> }*

The <signal definition > is defined in § 2.5.4, <signal list definition> in § 2.5.5, <view definition> in § 2.6.1.2, <variable definition> in § 2.6.1.1, <procedure area> in § 2.4.5, <timer definition> in § 2.8, <macro definition> and <macro diagram> in § 4.2.2, <import definition> in § 4.1.3, <select definition> in § 4.3.3, <data definition> in § 5.5.1, <start area> in § 2.6.2, <state area> in § 2.6.3, <in-connector area> in § 2.6.6, and <service interaction area> in § 4.10.1

An example of <process diagram> is shown in Figure 2.9.10 § 2.9.

Semantics

A process definition introduces the type of a process which is intended to represent a dynamic behaviour.

In the *Number-of-instances* the first value represents the *Number-of-instances* of the process which exist when the system is created, the second value represents the maximum number of simultanous instances of the process type.

A process instance is a communicating extended finite state machine performing a certain set of actions, denoted as transitions, accordingly to the reception of a given signal, whenever it is in a state. The completion of the transition results in the process waiting in another state, which is not necessarily different from the first one.

The concept of finite state machine has been extended in that the state resulting after a transition, besides the signal originating the transition, may be affected by decisions taken upon variables known to the process.

Several instances of the same process type may exist at the same time and execute asynchronously and in parallel with each other, and with other instances of different process type in the system.

When a system is created, the initial processes are created in a random order. The signal communication between the processes commences only when all the initial processes have been created.The formal parameters of these initial processes are initialized to an undefined value.

Process instances exist from the time that a system is created or can be created by create request actions which start the processes being interpreted; their interpretation start when the start action is interpreted; they may cease to exist by performing stop actions.

Signals received by process instances are denoted as input signals, and signals sent to process instances are denoted as output signals.

Signals may be consumed by a process instance only when it is in a state. The complete valid input signal set is the union of the set of signals in all signal routes leading to the process, the <valid input signal set>, the implicit signals and timer signals.

One and only one input port is associated with each process instance. When an input signal arrives at the process, it is put into the input port of the process instance

The process is either waiting in a state or active performing a transition. For each state, there is a save signal set (see also § 2.6.5). When waiting in a state, the first input signal whose identifier is not in the save signal set is taken from the queue and consumed by the process.

The input port may retain any number of input signals, so that several input signals are queued for the process instance. The set of retained signals are ordered in the queue according to their arrival time. If two or more signals arrive on different paths "simultaneously", they are arbitrarily ordered.

When the process is created, it is given an empty input port, and local variables are created with values assigned to them.

The formal parameters are variables which are created either when the system is created (but no actual parameters are passed to and therefore they are not initialized) or when the process instance is dynamically created.

To all process instances four expressions yielding a PId (see § 5.6.10) value may be used: SELF, PARENT, OFFSPRING and SENDER. They give a result for:

a) the process instance (SELF);

b) the creating process instance (PARENT);

c) the most recent process instance created by the process (OFFSPRING);

d) the process instance from which the last input signal has been consumed (SENDER) (see also § 2.6.4).

These expressions are further explained in § 5.5.4.3

SELF, PARENT, OFFSPRING and SENDER can be used in expressions inside the process instances.

For all process instances present at system initialization, the predefined PARENT expression always has the value NULL.

For all newly created process instances the predefined SENDER and OFFSPRING expressions have the value NULL.

2.4.5 *Procedure*

Procedures are defined by means of procedure definitions. The procedure is invoked by means of a procedure call referencing the procedure definition. Parameters are associated with a procedure call: these are used both to pass values, and also to control the scope of variables for the procedure execution. Which variables are affected by the interpretation of a procedure is controlled by the parameter passing mechanism.

Abstract grammar

Procedure-definition	::	*Procedure-name* *Procedure-formal-parameter** *Procedure-definition*-**set** *Data-type-definition* *Syn-type-definition*-**set** *Variable-definition*-**set** *Procedure-graph*

Procedure-name = *Name*

Procedure-formal-parameter = *In-parameter* |
Inout-parameter

In-parameter :: *Variable-name*
Sort-reference-identifier

Inout-parameter :: *Variable-name*
Sort-reference-identifier

Procedure-graph :: *Procedure-start-node*
State-node-**set**

Procedure-start-node :: *Transition*

Concrete textual grammar

<procedure definition> ::=
PROCEDURE {<procedure identifier>| <procedure name> } <end>
[<procedure formal parameters> <end>]
{ <data definition >
| <variable definition>
| <textual procedure reference>
| <procedure definition>
| <select definition>
| <macro definition> }*
<procedure body>
ENDPROCEDURE [<procedure name>|<procedure identifier>] <end>

<procedure formal parameters> ::=
FPAR <formal variable parameter>
{, <formal variable parameter> }*
<formal variable parameter> ::=
[IN/OUT
| IN]
<variable name> {, <variable name>}* <sort>
< procedure body> ::=
<process body>

The <variable definition> is defined in § 2.6.1.1, <textual procedure reference> in § 2.4.4, <macro definition> is defined in § 4.2, <select definition> in § 4.3.3, <data definition>in § 5.5.1, <sort> in § 5.2.2.

In a <procedure definition>, <variable definition> cannot contain REVEALED, EXPORTED, REVEALED EXPORTED, EXPORTED REVEALED <variable name>s (see § 2.6.1)
An example of <procedure definition> is shown in Figure 2.9.11.

Concrete graphical grammar

<procedure diagram> ::=
<frame symbol> **contains** {<procedure heading>
{ {<procedure text area>
| <procedure area>
| <macro diagram> }*
<procedure graph area> }**set** }

<procedure area> ::=
<graphical procedure reference>
| <procedure diagram>

<procedure text area> ::=
<text symbol> **contains**
{<variable definition>
| <data definition>
| <select definition>
| <macro definition> }*

<graphical procedure reference> ::=
<procedure symbol> **contains** <procedure name>

<procedure symbol> ::=

<procedure heading> ::=
PROCEDURE {<procedure name>| <procedure identifier> }
[<procedure formal parameters>]

<procedure graph area> ::=
<procedure start area>
{<state area> | <in-connector area> }*

<procedure start area> ::=
<procedure start symbol> **is followed by** <transition area>

<procedure start symbol> ::=

The <variable definition> is defined in § 2.6.1.1, <transition area> in § 2.6.7.1, <state area> in § 2.6.3, <in-connector area> in § 2.6.6, <macro definition> and <macro diagram> are defined in § 4.2, <select definition> in § 4.3.3, <data definition>in § 5.5.1.

An example of <procedure diagram> is shown in Figure 2.9.12 in § 2.9.

Semantics

A procedure is a means of giving a name to an assembly of items and representing this assembly by a single reference. The rules for procedures impose a discipline upon the way, in which the

assembly of items is chosen, and limit the scope of the name of variables defined in the procedure.
A procedure variable is a local variable within the procedure that can neither be revealed nor viewed, nor exported, nor imported. It is created when the procedure start node is interpreted, and it ceases to exist when the return node of the procedure graph is interpreted.

When a procedure definition is interpreted, its procedure graph is interpreted.

A procedure definition is interpreted only when a process instance calls it, and is interpreted by that process instance.

The interpretation of a procedure definition causes the creation of a procedure instance and the interpretation to commence in the following way:

a) A local variable is created for each *In-parameter*, having the *Name* and *Sort* of the *In-parameter*. The variable is assigned the value of the expression given by the corresponding actual parameter, which may be undefined.

b) If an actual parameter is empty the corresponding formal parameter is given the value undefined.

c) A formal parameter with no explicit attribute, has an implicit IN attribute.

d) A local variable is created for each *Variable-definition* in the *Procedure-definition*, having the *Name* and *Sort* of the *Variable-definition* .

e) Each *Inout-parameter* denotes a synonym name for the variable which is given in the actual parameter expression. This synonym name is used throughout the interpretation of the *Procedure-graph* when referring to the value of the variable or when assigning a new value to the variable.

f) The *Transition* contained in the *Procedure-start-node* is interpreted.

2.5 *Communication*

2.5.1 *Channel*

Abstract grammar

Channel-definition	::	*Channel-name* *Channel-path* [*Channel-path*]
Channel-path	::	*Originating-block* *Destination-block* *Signal-identifier*-**set**
Originating-block	=	*Block-identifier* | ENVIRONMENT
Destination-block	=	*Block-identifier* | ENVIRONMENT
Block-identifier	=	*Identifier*

Signal-identifier = *Identifier*
Channel-name = *Name*

The *Signal-identifier*-**set** must contain the list of all signals that may be conveyed on the channel-path(s) defined by the channel.

At least one of the end points of the channel must be a block.
If both end points are blocks, the blocks must be different.

The block end point(s) must be defined in the same scope unit as the channel is defined.

Concrete textual grammar

<channel definition> ::=
CHANNEL <channel name>
<channel path>
[<channel path>]
[<channel substructure definition>
| <textual channel substructure reference>]
ENDCHANNEL [<channel name>] <end>

<channel path> ::=
{ FROM <block identifier> TO <block identifier>
| FROM <block identifier> TO ENV
| FROM ENV TO <block identifier> }
WITH <signal list> <end>

The <signal list> is defined in § 2.5.5, <channel substructure definition> and <textual channel substructure reference> in § 3.2.3.

Where two <channel path>s are defined one must be in the reverse direction to the other.

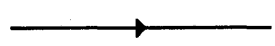
Concrete graphical grammar

<channel definition area> ::=
<channel symbol>
is associated with {<channel name>
{ [{<channel identifier> | <block identifier>}]
<signal list area> [<signal list area>]}**set** }
is connected to { <block area> {<block area> | <frame symbol>}
[<channel substructure association area>] }**set**

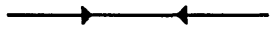
The <channel identifier> identifies an external channel connected to the <block substructure diagram> delimited by the <frame symbol>. The <block identifier> identifies an external block being a channel endpoint for the <channel substructure diagram> delimited by the <frame symbol>.

<channel symbol> ::=
<channel symbol 1>
| <channel symbol 2>
| <channel symbol 3>

<channel symbol 1> ::=



<channel symbol 2> ::=



<channel symbol 3> ::=

The <signal list area> is defined in § 2.5.5, <block area> and <frame symbol> in § 2.4.1, <channel substructure association area> in § 3.2.3.

For each arrowhead on the <channel symbol>, there must be a <signal list area>. A <signal list area> must be unambiguously close enough to the arrowhead to which it is associated.

Semantics

A channel represents a transportation route for signals. A channel can be considered as one or two independent unidirectional channel paths between two blocks or between a block and its environment.

Signals conveyed by channels are delivered to the destination endpoint.

Signals are presented at the destination endpoint of a channel in the same order they have been presented at its origin point. If two or more signals are presented simultaneously to the channel, they are arbitrarily ordered.

A channel may delay the signals conveyed by the channel. That means that a First-In-First-Out (FIFO) delaying queue is associated with each direction in a channel. When a signal is presented to the channel, it is put into the delaying queue. After an indeterminant and non-constant time interval, the first signal instance in the queue is released and given to one of the channels or signal routes which is connected to the channel.

Several channels may exist between the same two endpoints. The same signal type can be conveyed on different channels.

2.5.2 *Signal route*

Abstract grammar

Signal-route-definition	::	*Signal-route-name* *Signal-route-path* [*Signal-route-path*]
Signal-route-path	::	*Originating-process* *Destination-process* *Signal-identifier*-set
Originating-process	=	*Process-identifier* | ENVIRONMENT
Destination-process	=	*Process-identifier* | ENVIRONMENT
Signal-route-name	=	*Name*

At least one of the end points of the *Signal-route-path* must be a process.

If both end points are processes, the *Process-identifier*s must be different.

The process endpoint(s) must be defined in the same scope unit as the signal route is defined.

Concrete textual grammar

<signal route definition> ::=
SIGNALROUTE <signal route name>
<signal route path>
[<signal route path>]

<signal route path>::=
{ FROM <process identifier> TO <process identifier>
| FROM <process identifier> TO ENV
| FROM ENV TO <process identifier> }
WITH <signal list> <end>

The <signal list> is defined in § 2.5.5.

Where two <signal route path>s are defined one must be in the reverse direction to the other.

Concrete graphical grammar

<signal route definition area> ::=
<signal route symbol>
is associated with {<signal route name>
{[<channel identifier>] <signal list area> [<signal list area>] }**set** }
is connected to
{<process area> {<process area> | <frame symbol>} }**set**

<signal route symbol> ::=
<signal route symbol 1> | <signal route symbol 2>

<signal route symbol 1> ::=

<signal route symbol 2> ::=

A signal route symbol includes an arrowhead at one end (one direction) or one arrowhead at each end (bidirectional) to show the direction of the flow of signals.

For each arrowhead on the <signal route symbol>, there must be a <signal list area>. A <signal list area> must be unambiguously close enough to the arrowhead to which it is associated.

When the <signal route symbol> is connected to the <frame symbol>, then the <channel identifier> identifies a channel to which the signal route is connected.

Semantics

A signal route represents a transportation route for signals. A signal route can be considered as one or two independent unidirectional signal route paths between two processes or between a process and its environment.

Signals conveyed by signal routes are delivered to the destination endpoint.

A signal route does not introduce any delay in conveying the signals.

No signal route connects process instances of the same type. In this case, interpretation of the *Output-node* implies that the signal is put directly in the input port of the destination process.

Several signal routes may exist between the same two endpoints. The same signal type can be conveyed on different signal routes.

Model

A <valid input signal set> contains signals that the process is allowed to receive. A <valid input signal set>, however, must not contain timer signals. If a <block definition> contains <signal route definition>s then the <valid input signal set>, if any, need not contain signals in signal routes leading to the process.

If a <block definition> contains no <signal route definition>s, then all <process definition>s in that <block definition> must contain a <valid input signal set>. In that case the <signal route definition>s and the <channel to route connection>s are derived from the <valid input signal set>s, <output>s and channels terminating at the blocks boundary. The signals corresponding to a given direction between two processes in the implied signal route is the intersection of the signals specified in the <valid input signal set> of the destination process and the signals mentioned in an output of the originating process. If one of the endpoints is the environment then the input set/output set for that endpoint is the signals conveyed by the channel in the given direction.

2.5.3 *Connection*

Abstract grammar

Channel-to-route-connection	::	*Channel-identifier* *Signal-route-identifier*-**set**
Signal-route-identifier	=	*Identifier*

Other connect constructs are contained in § 3.

Each *Channel-identifier* connected to the enclosing block must be mentioned in exactly one*Channel-to-route-connection*.. The *Channel-identifier* in a *Channel-to-route-connection* must denote a channel connected to the enclosing block.

Each *Signal-route-identifier* in a *Channel-to-route-connection* must be defined in the same block as where the *Channel-to-route-connection* is defined and it must have the boundary of that block as one of its endpoints. Each *Signal-route-identifier* defined in the surrounding block and which has its environment as one of its endpoints, must be mentioned in one and only one *Channel-to-route-connection.*

For a given direction, the union of the *Signal-identifier* sets in the signal routes in a *Channel-to-route-connection* must be equal to the set of signals conveyed by the *Channel-identifier* in the same *Channel-to-route-connection* and corresponding to the same direction.

Concrete textual grammar

<channel to route connection> ::=
CONNECT <channel identifier>
AND <signal route identifier> {,<signal route identifier>}*
<end>

No <signal route identifier> in a <channel to route connection> may be mentioned twice.

Concrete graphical grammar

Graphically the connect construct is represented by the <channel identifier> associated to the signal route and contained in the <signal route definition area > (see § 2.5.2 *Concrete graphical grammar*).

2.5.4 *Signal*

Abstract grammar

Signal-definition	::	*Signal-name* *Sort-reference-identifier** [*Signal-refinement*]
Signal-name	=	*Name*

The *Sort-reference-identifier* is defined in § 5.2.2.

Concrete textual grammar

<signal definition>::=
SIGNAL {<signal name>[<sort list>][<signal refinement>] }
{,<signal name> [<sort list>] [<signal refinement>]}* <end>

<sort list> ::=
(<sort> {, <sort>}*)

<signal refinement> is defined in § 3.3, <sort> is defined in § 5.2.2.

Semantics

A signal instance is a flow of information between processes, and is an instantiation of a signal type defined by a signal definition. A signal instance can be sent by either the environment or a process and is always directed to either a process or the environment.

Two PId values (see § 5.6.10) denoting the origin and the destination processes, the <signal identifier> specified in the corresponding output, and other values, whose sorts are defined in the signal definition, are associated with each signal instance.

2.5.5 *Signal list definition*

A <signal list identifier> may be used in <channel definition>, <signal route definition>, <signal list definition>,<valid input signal set> and <savelist>, as a shorthand to list signal identifiers and timer signals.

Concrete textual grammar

<signal list definition> ::=
SIGNALLIST<signal list name>= <signal list><end>

<signal list> ::=
<signal item> { , <signal item>}*

<signal item> ::=

<signal identifier> | <priority signal identifier> | (<signal list identifier>)
| <timer identifier>

The <signal list> which is constructed by replacing all <signal list identifier>s in the list by the <signal identifier>s they denote, corresponds to a *Signal-identifier*-**set** in the *Abstract grammar*. In every such constructed <signal list>, every <signal identifier> must be distinct.

Concrete graphical grammar

<signal list area> ::=

<signal list symbol> **contains** <signal list>

<signal list symbol> ::=

2.6 *Behaviour*

2.6.1 *Variables*

2.6.1.1 *Variable definition*

Abstract grammar

Variable-definition	::	*Variable-name* *Sort-reference-identifier* [REVEALED]
Variable-name	=	*Name*

Concrete textual grammar

<variable definition> ::=

DCL [REVEALED| EXPORTED | REVEALED EXPORTED | EXPORTED REVEALED]
<variable name> {, <variable name>}* <sort> [:= <ground expression>]
{,<variable name>{, <variable name>}* <sort> [:=<ground expression>] }*<end>

Exported variable is defined in § 4.13.

Semantics

The semantics of variables is defined in § 2.3.2. The value of a variable can only be modified by its owner. The owner of a variable is the process (or procedure) where the variable is declared. The value of a variable is only known to its owner unless the variable has the REVEALED attribute. The REVEALED attribute allows all the other processes of the same block to view the variable, provided they have the view definition in declaring the variable.

Model

The <ground expression> in a <variable definition> or default value in a <sort> has no corresponding abstract syntax. It is derived syntax for specifying a sequence of assignment statements in the initial transition of the surrounding scope unit. The assignment statements assigns the <ground expression> to all the <variable name> mentioned in the <variable definition>. If both

a default value in a <sort> and a <ground expression> in the <variable definition >is specified, the <ground expression> in the <variable definition> applies.

2.6.1.2 *View definition*

Abstract grammar

View definition :: *Variable-identifier*
Sort-reference-identifier

The *Variable-definition* designated by *Variable-identifier* must have the REVEALED attribute, and it must be of the same sort as the *Sort-reference-identifier* denoted.

Concrete textual grammar

<view definition> ::=
VIEWED
<variable identifier> {, <variable identifier>}* <sort>
{,<variable identifier>{, <variable identifier>}*<sort>}* <end>

The qualifier in <variable identifier> in <view definition> may be omitted only if there exist one and only one <process definition> in the block, which have a <variable definition> defining a <variable name> which is the same as the <variable name> mentioned in <view definition> and which have the REVEALED attribute, and which is of the same <sort> as denoted by the <sort> in the <view definition>.

Semantics

The view mechanism allows a process instance to see the viewed variable value continuously as if it were locally defined. The viewing process instance doesn't however have any right to modify it.

2.6.2 *Start*

Abstract grammar

Process-start-node :: *Transition*

Concrete textual grammar

<start> ::=
START <end> <transition>

Concrete graphical grammar

<start area> ::=
<start symbol> **is followed by** <transition area>

<start symbol> ::=

Semantics

The *Transition* of the *Process-start-node* is interpreted.

2.6.3 *State*

Abstract grammar

State-node	::	*State-name* *Save-signalset* *Input-node*-**set**
State-name	=	*Name*

State-node s within a *process-graph* respectively *procedure-graph* have different *State-name*s.

For each *State-node*, all *Signal-identifiers* (in the complete valid input signal set) appear in either a *Save-signalset* or an *Input-node*.

The *Signal-identifier* s in the *Input-node*-**set** must be distinct.

Concrete textual grammar

```
<state> ::=
            STATE <state list> <end>
                {<input part>
            |   <priority input>
            |   <save part>
            |   <continuous signal>}*
            [ENDSTATE [<state name>] <end>]

<state list> ::=
            {<state name> { , <state name> }*}
        |   <asterisk state list>
```

The <input part> is defined in § 2.6.4, <save part> in § 2.6.5, <continuous signal> in § 4.11, <asterisk state list> in § 4.4 and <priority input> in § 4.10.2.

When the <state list> contains one <state name> then the <state name> represents a *State-node*. For each *State-node*, the *Save-signalset* is represented by the <save part> and any implicit signal saves. For each *State-node* , the *Input-node* **set** is represented by the <input part> and any implicit input signals.

The optional <state name> ending a <state> may be specified only if the <state list> in the <state> consists of a single <state name> in which case it must be the same <state name> as in the <state list>.

Concrete graphical grammar

<state area> ::=
<state symbol> **contains** <state list> **is associated with**
{<input association area>
| <priority input association area>
| <continuous signal association area>
| <save association area> }*

<state symbol> ::=

<input association area> ::=
<solid association symbol> **is connected to** <input area>

<save association area> ::=
<solid association symbol> **is connected to** <save area>

The <input area> is defined in § 2.6.4, <save area> in § 2.6.5, <continuous signal association area> in § 4.11, <priority input association area> in § 4.10.2.

A <state area> represents one or more *State-node*s.

The <solid association symbol>s originating from a <state symbol> may have a common originating path.

Semantics

A state represents a particular condition in which a process instance can consume a signal instance resulting in a transition. If there are no retained signal instances then the process waits in the state until a signal instance is received.

Model

When the <state list> of a certain <state> contains more than one <state name>s, a copy of the <state> is created for each such <state name>. Then the <state> is replaced by these copies.

2.6.4 *Input*

Abstract grammar

Input-node	::	*Signal-identifier* [*Variable-identifier*]* *Transition*
Variable-identifier	=	*Identifier*

The length of the [*Variable-identifier*]* must be the same as the number of *Sort-reference-identifier*s in the *Signal-definition* denoted by the *Signal-identifier*.

The sorts of the variables should correspond by position to the sorts of the values that can be carried by the signal.

It is not allowed to specify more variables to receive than the number of values conveyed by the signal instance.

Concrete textual grammar

<input part> ::=
INPUT <input list> <end>
[<enabling condition>]<transition>

<input list> ::=

<asterisk input list>
| <stimulus> { ,<stimulus> }*

<stimulus> ::=
{ <signal identifier>
| <timer identifier>} [([<variable identifier>] {,[<variable identifier>]}*)]

The <transition> is defined in § 2.6.7, <enabling condition> in § 4.12, and <asterisk input list> in § 4.6.

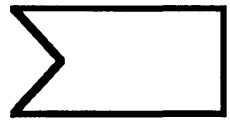
When the <input list > contains one <stimulus>, then the <input part> represents an <input node>. In the *Abstract grammar*, timer signals (<timer identifier>) are also represented by *Signal-identifier*. Timer signals and ordinary signals are distinguished only where appropriate, as in many respects they have similar properties. The exact properties of timer signals are defined in § 2.8.

A <transition> must have a transition terminator as defined in § 2.6.7.2

Concrete graphical grammar

<input area> ::=
<input symbol> **contains** <input list>
is followed by {[<enabling condition area>] <transition area>}

<input symbol> ::=



The <transition area> is defined in § 2.6.7, <enabling condition area> in § 4.12.

An <input area> whose <input list> contains one <stimulus> corresponds to one *Input-node*. Each of the <signal identifiers> contained in an <input symbol> gives the name of one of the *Input-nodes* which this <input symbol> represents.

Semantics

An input allows the consumption of the specified input signal instance. The consumption of the input signal instance makes the information conveyed by the signal available to the process. The variables associated with the input are assigned the values conveyed by the consumed signal. If there is no variable associated with the input for a sort specified in the signal, the value of this sort is discarded.

The SENDER expression of the consuming process is given the PId value of the originating process instance, carried by the signal instance.

Signal instances flowing from the environment to a process instance within the system will always have a PId value different from any in the system. This is accessed using the SENDER expression.

Model

When the <stimulus>s list of a certain <input part> contains more than one <stimulus>, a copy of the <input part> is created for each such <stimulus>. Then the <input part> is replaced by these copies.

2.6.5 *Save*

A save specifies a set of signal identifiers whose instances are not relevant to the process in the state to which the save is attached, and which need to be saved for future processing.

Abstract grammar

Save-signalset :: *Signal-identifier*-**set**

In each *State-node* the *Signal-identifier*s contained in the *Save-signalset* must be different.

Concrete textual grammar

<save part> ::=
SAVE <save list> <end>
<save list> ::=
{<signal list> | <asterisk save list>}

A <save list> represents the *Signal-identifier*-**set**. The <asterisk save list> is a shorthand notation explained in § 4.8.

Concrete graphical grammar

<save area> ::=

<save symbol> **contains** <save list>

<save symbol> ::=

Semantics

The saved signals are retained in the input port in the order of their arrival.

The effect of the save is valid only for the state to which the save is attached. In the following state, signal instances that have been "saved" are treated as normal signal instances.

2.6.6 *Label*

Concrete textual grammar

<label> ::=
<connector name>:

All the <connector name>s defined in a <process body> must be distinct.

A label represents the entry point of a "jump" from the corresponding join statements with the same <connector name>s in the same <process body>.

"Jumps" are only allowed to labels within the same <process body>.

Concrete graphical grammar

<in-connector area> ::=
<in-connector symbol> **contains** <connector name> **is followed by** <transition area>

<in-connector symbol> ::=

<transition area> is defined in § 2.6.7.1.

An <in-connector area> represents the continuation of a <flow line symbol> from a corresponding <out-connector area> with the same <connector name> in the same <process graph area> or <procedure graph area>.

2.6.7 *Transition*

2.6.7.1 *Transition body*

Abstract grammar

Transition	::	*Graph-node* * (*Terminator* | *Decision-node*)
Graph-node	::	*Task-node* | *Output -node* | *Create-Request-node* | *Call-node* | *Set-node* | *Reset-node* |
Terminator	::	*Nextstate-node* | *Stop-node* | *Return-node*

Concrete textual grammar

```
<transition> ::=
        {<transition string> [<terminator statement>] }
    |   <terminator statement>

<transition string> ::=
        {<action statement>}+

<action statement> ::=
        [<label>] <action> <end>

<action> ::=
        <task>
    |   <output>
    |   <priority output>
    |   <create request>
    |   <decision>
    |   <transition option>
    |   <set>
    |   <reset>
    |   <export>
    |   <procedure call>

<terminator statement> ::=
        [<label>] <terminator> <end>

<terminator> ::=
        <nextstate>
    |   <join>
```

| <stop>
| <return>

The <task> is defined in § 2.7.1, <output> in § 2.7.4, <create request> in § 2.7.2, <decision> in § 2.7.5, <set> and <reset> in § 2.8, <procedure call> in § 2.7.3, <nextstate> in § 2.6.7.2.1, <join> in § 2.6.7.2.2, <stop> in § 2.6.7.2.3, <return> in § 2.6.7.2.4, <priority output> in § 4.10.2,<transition option> in § 4.3.4, and <export> in § 4.13.

If the <terminator> of a <transition> is omitted then the last action in the <transition> must contain a terminating <decision> (see § 2.7.5) or terminating <transition option>, except for all <transition>s contained in <decision>s and <transition option>s (<transition option> is defined in § 4.3.4)

No <terminator> or <action> may follow a <terminator>, a terminating <transition option> or a terminating <decision>.

Concrete graphical grammar

<transition area> ::=
[<transition string area>] **is followed by**
{<state area>
| <nextstate area>
| <decision area>
| <stop symbol>
| <merge area>
| <out connector area>
| <return symbol>
| <transition option area> }

<transition string area> ::=
{<task area>
| <output area>
| <priority output area>
| <set area>
| <reset area>
| <export area>
| <create request area>
| <procedure call area> }
[**is followed by** <transition string area>]

The <task area> is defined in § 2.7.1, <output area> in § 2.7.4, <create request area> in § 2.7.2, <decision area> in § 2.7.5, <set area> and <reset area> in § 2.8, <procedure call area> in § 2.7.3, <nextstate area> in § 2.6.7.2.1, <merge area> in § 2.6.7.2.2, <stop symbol> in § 2.6.7.2.3, <return symbol> in § 2.6.7.2.4, <priority output area> in § 4.10.2,<transition option area> in § 4.3.4, <export area> in § 4.13, and <out-connector area> in § 2.6.7.2.2.

A transition consists of a sequence of actions to be performed by the process.

The <transition area> corresponds to *Transition* and <transition string area> corresponds to *Graph-node**.

Semantics

A transition performs a sequence of actions. During a transition, the data of a process may be manipulated and signals may be output. The transition will end with the process entering a state, with a stop or with a return.

2.6.7.2 *Transition terminator*

2.6.7.2.1 *Nextstate*

Abstract grammar

Nextstate-node :: *State-name*

The *State-name* specified in a nextstate must be the name of a state within the same *Process-graph* or *Procedure-graph.*

Concrete textual grammar

<nextstate>::=
NEXTSTATE <nextstate body>

<nextstate body> ::=
{<state name>|<dash nextstate>}

<dash nextstate> is defined in § 4.9.

Concrete graphical grammar

<nextstate area> ::=
<state symbol> **contains** <nextstate body>

Semantics

A nextstate represents a terminator of a transition. It specifies the state the process instance will assume when terminating the transition.

2.6.7.2.2 *Join*

A join alters the flow in a <process diagram> or <process body> by expressing that the next <action statement> to be interpreted is the one which contains the same <connector name>.

Concrete textual grammar

<join> ::=
JOIN <connector name>

There must be one and only one <connector name> corresponding to a <join> within the same <process body>, <procedure body> respectively <service body>.

Concrete graphical grammar

<merge area> ::=
<merge symbol> **is connected to** <flow line symbol>

<merge symbol> ::=
<flow line symbol>

<flow line symbol> ::=
———————

<out-connector area> ::=
<out-connector symbol> **contains** <connector name>

<out-connector symbol> ::=
<in-connector symbol>

For each <out-connector area> in a <process graph area> or <procedure graph area> there must be one and only one <in-connector area> respectively in that <process graph area> or <procedure graph area> with the same <connector name>

An <out-connector area> corresponds to a <join> in the *Concrete textual grammar*. If a <merge area> is included in a <transition area> it is equivalent to specifying an <out-connector area> in the <transition area> which contains a unique <connector name> and attaching an <in-connector area>, with the same <connector name> to the <flow line symbol> in the <merge area>.

Model

In the abstract syntax a <join> or <out-connector area> is derived from the <transition string> wherein the first <action statement> or area has the same <connector name> attached.

2.6.7.2.3 *Stop*

Abstract grammar

Stop-node :: ()

A *Stop-node* must not be contained in a *Procedure-graph.*

Concrete textual grammar

<stop> ::=
STOP

Concrete graphical grammar

<stop symbol> ::=

Semantics

The stop causes the immediate halting of the process instance issuing it. This means that the retained signals in the input port are discarded and that the variables and timers created for the process, the input port and the process will cease to exist.

2.6.7.2.4 *Return*

Abstract grammar

Return-node :: *()*

A *Return-node* must not be contained in a *Process-graph.*

Concrete textual grammar

<return> ::=
RETURN

Concrete graphical grammar

<return symbol> ::=

Semantics

A *Return-node* is interpreted in the following way:

a) All variables created by the interpretation of the Procedure-start-node will cease to exist.

b) Interpreting the *Return-node* completes the interpretation of the *Procedure-graph* and the procedure instance ceases to exist.

c) Hereafter the calling process (or procedure) interpretation continues at the node following the call.

2.7 *Action*

2.7.1 *Task*

Abstract grammar

Task-node :: *Assignment-statement* |
Informal -text

Concrete textual grammar

<task> ::=
TASK <task body>

<task body> ::=
{<assignment statement>{,<assignment statement>}*}
| {<informal text> {,<informal text>}*}

<assignment statement> is defined in § 5.5.3

Concrete graphical grammar

<task area> ::=
<task symbol> **contains** <task body>

<task symbol> ::=

Semantics

The interpretation of a *Task-node* is the interpretation of the *Assignment-statement* which is explained in § 5.5.3, or the interpretation of the *Informal-text* which is explained in § 2.2.3

Model

A <task> and a <task area> may contain several <assignment statement>s or <informal text>. In that case it is derived syntax for specifying a sequence of <task>s, one for each <assignment statement> or <informal text> such that the original order they were specified in the <task body> is retained.

This shorthand is expanded before any <import expression> is expanded (see §4.13).

2.7.2 *Create*

Abstract grammar

Create-request-node :: *Process-identifier*
[*Expression*]*

Process-identifier = *Identifier*

The number of *Expressions* in the [*Expression*]* must be the same as the number of *Process-formal-parameters* in the *Process-definition* of the *Process-identifier*. Each *Expression* must have the same sort as the corresponding by position *Process-formal-parameter* in the *Process-definition* denoted *Process-identifier*.

Concrete textual grammar

<create request> ::=
CREATE <create body>

<create body> ::=
<process identifier> [<actual parameters>]

<actual parameters> ::=
([<expression>] {,[<expression>]}*)

<expression> is defined in § 5.

Concrete graphical grammar

<create request area> ::=
<create request symbol> **contains** <create body>

<create request symbol> ::=

A <create request area> represents a *Create-request-node*.

Semantics

When a process instance is created, it is given an empty input port, variables are created and the actual parameter expressions are interpreted in the order given, and assigned (as defined in § 5.5.3)

to the corresponding formal parameters. If an actual parameter is empty, the corresponding formal parameter is given the value undefined.Then the process starts by interpreting the start node in the process graph.

The created process then executes asynchronously and in parallel with other processes.

The create action causes the creation of a process instance in the same block. The created process PARENT has the same PId value as the creating process SELF. The created process SELF and the creating process OFFSPRING expressions both have a newly created PId value (see § 5.6.10.1).

If an attempt is made to create more process instances than specified by the maximum number of instances in the process definition, then no new instance is created, the OFFSPRING expression of the creating process has the value NULL and interpretation continues.

2.7.3 *Procedure Call*

Abstract grammar

Call-node :: *Procedure-identifier*
[*Expression*] *

Procedure-identifier = *Identifier*

The length of the [*Expression*]* must be the same as the number of the *Procedure-formal-parameters* in the *Procedure-definition* of the *Procedure-identifier.*

Each *Expression* corresponding by position to an IN *Process-formal-parameter* must have the same sort as the *Process-formal-parameter.*

Each *Expression* corresponding by position to an IN/OUT *Process-formal-parameter* must be a *Variable-identifier* with the same *Sort-reference-identifier* as the *Process-formal-parameter.*

There must be an *Expression* for each IN/OUT *Process-formal-parameter.*

Concrete textual grammar

<procedure call> ::=
CALL <procedure call body>

<procedure call body> ::=
<procedure identifier> [<actual parameters>]

<actual parameters> are defined in 2.7.2.

An example of <procedure call> is given in Figure 2.9.13 in § 2.9.

Concrete graphical grammar

<procedure call area> ::=
<procedure call symbol> **contains** <procedure call body>

<procedure call symbol> ::=

The <procedure call area> represents the *Call-node*.

An example of <procedure call area> is shown in Figure 2.9.14 in § 2.9.

Semantics

The interpretation of a procedure call node transfers the interpretation to the procedure definition referenced in the call node, and that procedure graph is interpreted. The node of the procedure graph are interpreted in the same manner as the equivalent nodes of a process graph.

The interpretation of the calling process is suspended until the interpretation of the called procedure is finished.

The actual parameter expressions are interpreted in the order given.

A special semantics is needed as far as data and parameters interpretation is concerned (the explanation is contained in § 2.4.4).

2.7.4 *Output*

Abstract grammar

Output-node	::	*Signal-identifier* [*Expression*]* [*Signal-destination*] *Direct-via*
Signal-destination	=	*Expression*
Direct-via	=	*Signal-route-identifier*-**set**

The length of the [*Expression*]* must be the same as the number of *Sort-reference-identifiers* in the *Signal-definition* denoted by the *Signal-identifier*.

Each *Expression* must have the same sort as the corresponding (by position) *Sort-identifier-reference* in the *Signal-definition*.

For every possible consistent subset (see § 3) there must exist at least one communication path (either implicit to own process type, or explicit via signal routes and possibly channels) to the environment or to a process type having *Signal-identifier* in its valid input signal set and originating from the process type where the *Output-node* is used.

For each *Signal-route-identifier* in *Direct-via* it must hold that the *Originating-process* in (one of) the *Signal-route-path*(s) in the signal route must be of the same process type as the process containing the *Output-node* and the *Signal-route-path* must include *Signal-identifier* in its set of *Signal-identifiers*.

If no *Signal-route-identifier* is specified in *Direct-via*, any process, for which there exists a communication path, may receive the signal.

Concrete textual grammar

<output> ::=
OUTPUT <output body>

<output body> ::=
<signal identifier>
[<actual parameters>]{, <signal identifier> [<actual parameters>]}*
[TO <PId expression>]
[VIA {<signal route identifier>{,<signal route identifier>}*
| {<channel identifier>{,<channel identifier>}* }]

The <actual parameters> are defined in § 2.7.2, <expression> in § 5.4.2.1.

It is not allowed to specify a <channel identifier> in the VIA construct if any signal routes are specified for the block.

For each <channel identifier> in an <output> there must exist a channel originating from the enclosing block, and able to convey the signals denoted by the <signal identifier>s contained in the <output>.

The TO <PId expression> represents the *Signal-destination.*

The VIA construct represents the *Direct-via* .

Concrete graphical grammar

<output area> ::=
<output symbol> **contains** <output body>

<output symbol> ::=

Semantics

The *Signal-destination* PId expression is interpreted after other expressions in the *Output-node.*

The values conveyed by the signal instance are the values of the actual parameters in the output. If there is no actual parameter in the output for a sort in the signal definition, the undefined value is conveyed by the signal.

The origin PId value conveyed by the signal instance is the value associated with SELF (of the

process performing the output action). The destination PId value conveyed by the signal instance is the value of the signal destination PId expression contained in the output.

The signal instance is then delivered to a communication path able to convey it to the specified destination process instance.

If no *Signal-destination* is specified, then there must exist one and only one receiver which may receive the signal according to the signal routes or channels specified in *Direct-via*. The destination PId value implicitly conveyed by the signal instance is the PId value of this receiver.

The environment may always receive any signal in the signal set of a channel which lead to the environment.

Note that specifying the same *channel identifier* or *signal route identifier* in the *Direct-via* of two *Output-nodes* does not automatically mean that the signals are queued in the input port in the same order as the *Output-nodes* are interpreted. However, order is preserved if the two signals are conveyed by identical channels connecting the *Originating-process* with the *Destination-process* or if the processes are defined within the same block.

If a syntype is specified in the signal definition and an expression is specified in the output, then the range check defined in § 5.4.1.9.1 is applied to the expression. If the range check is equivalent to False then the output is in error and the future behaviour of the system is undefined.

An output sent to a non existent process instance (or no longer existent) causes an interpretation error. The evaluation on the existence of a process instance is made at the same time the output is interpreted. A subsequent stopping of the receiving process instance causes the discarding of the signal from the input port and no error condition is reported.

Model

If several pairs of (<signal identifier> <actual parameters>) are specified in an <output body> it is derived syntax for specifying a sequence of <output>s or <output area>s in the same order specified in the original <output body> each containing a single pair of (<signal identifier> <actual parameters>). The TO clause and the VIA clause are repeated in each of the <output>s or <output area>s. This shorthand is expanded before any shorthands in the contained expressions are expanded.

2.7.5 *Decision*

Abstract grammar

Decision-node	::	*Decision-question* *Decision-answer*-set [*Else-answer*]
Decision-question	=	*Expression* | *Informal-text*
Decision-answer	::	*(Range-condition* | *Informal-text) Transition*
Else-answer	::	*Transition*

The *Decision-answers* must be mutually exclusive.

If the *Decision-question* is an *Expression*, the *Range-condition* of the *Decision-answers* must be of the same sort as the *Decision-question*.

Concrete textual grammar

<decision> ::=
DECISION <question> <end> <decision body> ENDDECISION

<decision body> ::=
{<answer part> <else part>}
| {<answer part> {<answer part>}$^{+}$ [<else part>] }

<answer part> ::=
(<answer>) : [<transition>]

<answer> ::=
<range condition> | <informal text>

<else part> ::=
ELSE:[<transition>]

<question> ::=
<question expression> | <informal text>

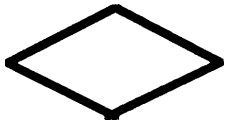
<range condition> is defined in § 5.4.1.9.1, <transition> in § 2.6.7.1, <informal text> in § 2.2.3.

A <decision> or <transition option> (defined in § 4.3.4) is terminating if each <answer part> and <else part> in the <decision body> contains a <transition> where a <terminator statement> is specified, or contains a <transition string> whose last <action statement> contains a terminating decision or option.

Concrete graphical grammar

<decision area> ::=
<decision symbol> **contains** <question>
is followed by
{ {<graphical answer part> <graphical else part>} **set**
| {<graphical answer part> {<graphical answer part>}+ [<graphical else part>] } **set** }

<decision symbol> ::=



<graphical answer part> ::=
<flow line symbol> **is associated with** <graphical answer>
is followed by <transition area>

<graphical answer> ::=
<answer> | (<answer>)

<graphical else part> ::=
<flow line symbol> **is associated with** ELSE
is followed by <transition area>

The <transition area> is defined in § 2.6.7.1 and <flow line symbol> in § 2.6.7.2.2.

The <graphical answer> and ELSE may be placed along the associated <flow line symbol>, or in the broken <flow line symbol>.

The <flow line symbol>s originating from a <decision symbol> may have a common originating path.

A <decision area> represents a *Decision-node*.

Semantics

A decision transfers the interpretation to the outgoing path whose range condition contains the value given by the interpretation of the question. A set of possible answers to the question is defined, each of them specifying the set of actions to be interpreted for that path choice.

One of the answers may be the complement of the others. This is achieved by specifying the *Else-answer*, which indicates the set of activities to be performed when the value of the expression on which the question is posed, is not covered by the values or set of values specified in the other answers.

Whenever the *Else-answer* is not specified, the value resulting from the evaluation of the question expression must match one of the answers.

There is syntactic ambiguity between <informal text> and <character string> in question> and <answer>. If the <question> and all <answer>s are <character string>, then all of these are interpreted as <informal text>. If the <question> or any <answer> is a <character string> which does not match the context of the decision, then the <character string> denotes <informal text>. The context of the decision (i.e. the sort) is determined without regard to <answer>s which are <character string>.

Model

If a <decision> is not a terminating decision then it is derived syntax for a <decision> wherein all the <answer part>s and the <else part> have inserted in their <transition> a <join> to the first <action statement> following the decision or, if the decision is the last <action statement> in a <transition string>, to the following <terminator statement>.

2.8 *Timer*

Abstract grammar

Timer-definition :: *Timer-name Sort-reference-identifier**

Timer-name	=	*Name*
Set-node	::	*Time-expression* *Timer-identifier* *Expression**
Reset-node	::	*Timer-identifier* *Expression**
Timer-identifier	=	*Identifier*
Time-expression	=	*Expression*

The sorts of the *Expression** in the *Set-node* and *Reset-node* must correspond by position to the *Sort-reference-identifier** directly following the *Timer-name* identified by the *Timer-identifier*.

The *Expressions* in a *Set-node* or *Reset-node* must be evaluated in the order given.

Concrete textual grammar

<timer definition> ::=
TIMER <timer name> [<sort list>]
{ , <timer name> [<sort list>] }* <end>

<reset> ::=
RESET (<reset statement> { , <reset statement> }*)

<reset statement> ::=
<timer identifier> [(<expression list>)]

<set> ::=
SET <set statement> { , <set statement> }*

<set statement> ::=
(<time expression> , <timer identifier> [(<expression list>)])

<sort list> and <expression list> are defined in § 2.5.4 and § 5.5.2.1 respectively.

A <reset statement> represents a *Reset-node*; a <set statement> represents a *Set-node*. If a <reset> contains several <reset statement>s, then they must be interpreted in the order given. If a <set> contains several <set statement>s, then they must be interpreted in the order given.

Concrete graphical grammar

<set area> ::=
<task symbol> **contains** <set>

<reset area> ::=
<task symbol> **contains** <reset>

Semantics

A timer instance is an object, owned by a process instance, that can be active or inactive. Two

occurrences of a timer identifier followed by an expression list refer to the same timer instance only if the two expression lists have the same values.

When an inactive timer is set, a time value is associated with the timer. Provided there is no reset or other setting of this timer before the system time reaches this time value, a signal with the same name as the timer is put in the input port of the process. The same action is taken if the timer is set to a time value minor than NOW. After consumption of a timer signal the SENDER expression yields the same value as the SELF expression. If an expression list is given when the timer is set, the values of these expression(s) are contained in the timer signal in the same order. A timer is active from the moment of setting up to the moment of consumption of the timer signal.

If a sort specified in a timer definition is a syntype, then the range check defined in § 5.4.19.1 applied to the corresponding expression in a set or reset must be True, otherwise the system is in error and the further behaviour of the system is undefined.

When an inactive timer is reset, it remains inactive.

When an active timer is reset, the association with the time value is lost, if there is a corresponding retained timer signal in the input port then it is removed, and the timer becomes inactive.

When an active timer is set, this is equivalent to resetting the timer, immediately followed by setting the timer. Between this reset and set the timer remains active.

Before the first setting of a timer instance it is inactive.

2.9 *Examples*

```
--------
INPUT S1 /*example*/;
TASK /* example*/ T1:=0;
--------
```

FIGURE 2.9.1

Example of comment (PR)

```
---------
INPUT I1 COMMENT 'example';
TASK T1:=0;
---------
```

FIGURE 2.9.2

Example of comment (PR)

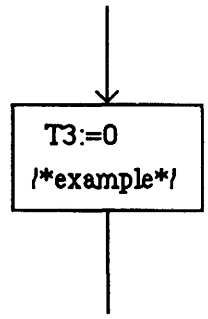


FIGURE 2.9.3

Example of comment (GR)

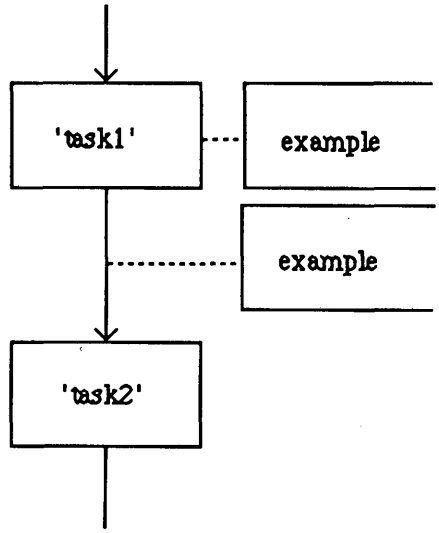


FIGURE 2.9.4

Example of comment (GR)

```
SYSTEM DAEMON_GAME;

/* This system is a game.....A player logs out by the signal Endgame */

SIGNAL Newgame, Probe, Result, Endgame, Gameid, Win, Lose, Score (Integer),
Subscr,Endsubscr, Bump;

CHANNEL C1
        FROM ENV TO Blockgame
        WITH Newgame, Probe, Result, Endgame;
        FROM Blockgame TO ENV
        WITH Gameid, Win, Lose, Score;
ENDCHANNEL C1;

CHANNEL C3 FROM Blockgame TO Blockdaemon
        WITH Subscr, Endsubscr;
ENDCHANNEL C3;

CHANNEL C4 FROM Blockdaemon TO Blockgame
        WITH Bump;
ENDCHANNEL C4;

BLOCK Blockgame REFERENCED;

BLOCK Blockdaemon REFERENCED;

ENDSYSTEM DAEMON_GAME;
```

FIGURE 2.9.5

Example of a system specification (PR)

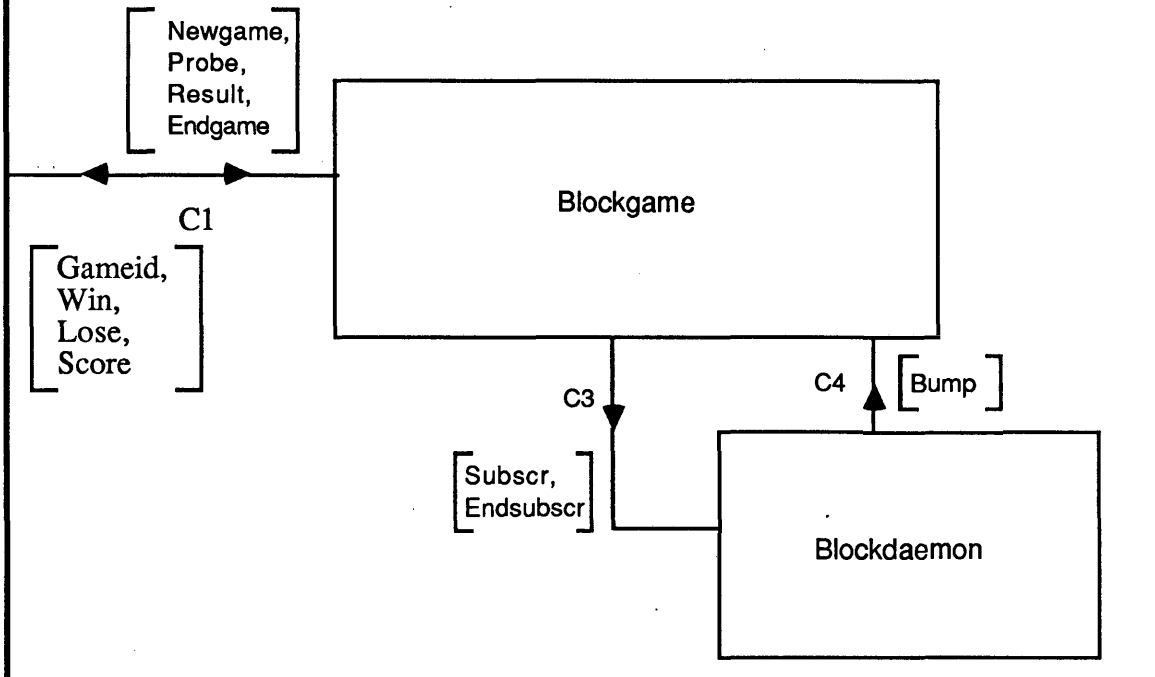
SYSTEM DAEMON_GAME 1(1)

```
/* This system is a game having any number of players. The players belong to the
   environment of the system. A "daemon" in the system produces Bump signals
   randomly. A player has to guess whether the number of the generated Bump
   signals is odd or even. The guess is made by sending a Probe signal to the
   system. The system replies by the signal Win if the number of the generated
Bump signals is odd, otherwise by the signal Lose.

   The system keeps track of the score of each player. A player can ask for the
   current value of his score by the signal Result, which is answered by the system
   with the signal Score.

   Before a player can start playing, he must  log in. This is accomplished by the
   signal Newgame. A player logs out by the signal Endgame.                    */

SIGNAL  Newgame, Probe, Result, Endgame, Gameid, Win, Lose, Score(integer),
        Subsrc, Endsubscr,  Bump;
```



T1003050-88

FIGURE 2.9.6
Example of a system specification (GR)

```
BLOCK Blockgame;

CONNECT C1 AND R1,R2,R3;
CONNECT C3 AND R4;
CONNECT C4 AND R5;
SIGNALROUTE R1 FROM ENV TO Monitor WITH Newgame;
SIGNALROUTE R2 FROM ENV TO Game
        WITH Probe, Result, Endgame;
SIGNALROUTE R3 FROM Game TO ENV
        WITH Gameid, Win, Lose, Score;
SIGNALROUTE R4 FROM Game TO ENV
        WITH Subscr, Endsubscr;
SIGNALROUTE R5 FROM ENV TO Game WITH Bump;

PROCESS Monitor (1,1) REFERENCED;

PROCESS Game (0,) REFERENCED;

ENDBLOCK Blockgame;
```

FIGURE 2.9.7

Example of block specification (PR)

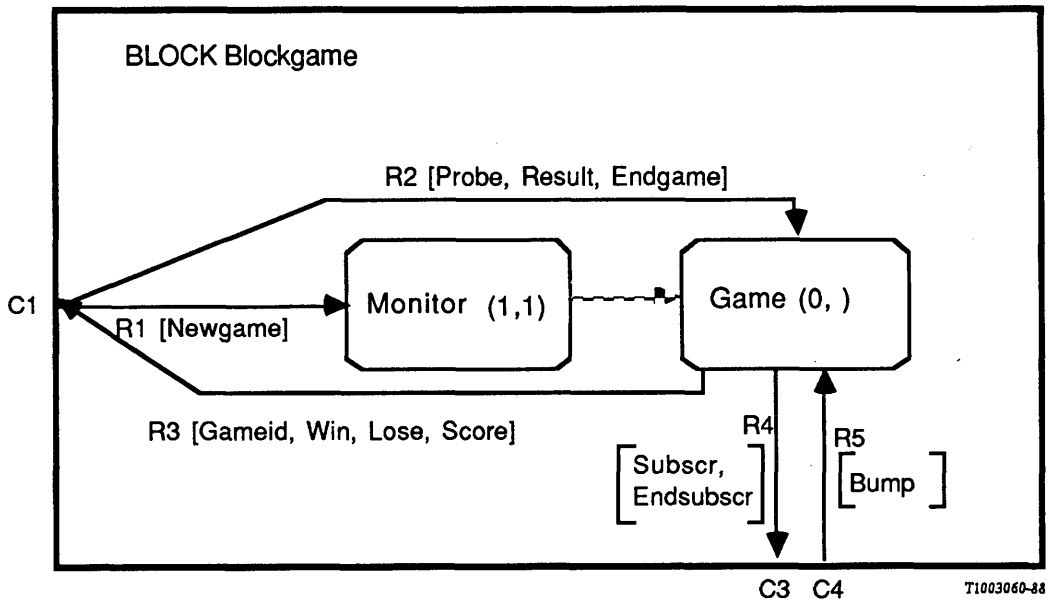


FIGURE 2.9.8

Example of a block diagram

```
PROCESS Game (0, );
FPAR Player Pid;

DCL
        Count Integer; /*counter to keep track of score */

START;

   OUTPUT Subscr;
   OUTPUT Gameid TO Player;
   TASK Count:=0;
NEXTSTATE Even;

STATE Even;

 INPUT Probe;
   OUTPUT Lose TO Player;
   TASK Count:=Count-1;
NEXTSTATE -;

 INPUT Bump;
NEXTSTATE Odd;

STATE Odd;

 INPUT Bump;
NEXTSTATE Even;

 INPUT Probe;
   OUTPUT Win TO Player;
   TASK Count:=Count+1;
NEXTSTATE -;

STATE *;

 INPUT Result;
   OUTPUT Score(Count) TO Player;
NEXTSTATE -;

 INPUT Endgame;
   OUTPUT Endsubscr;
STOP;

ENDPROCESS Game;
```

FIGURE 2.9.9

Example of process specification (PR)

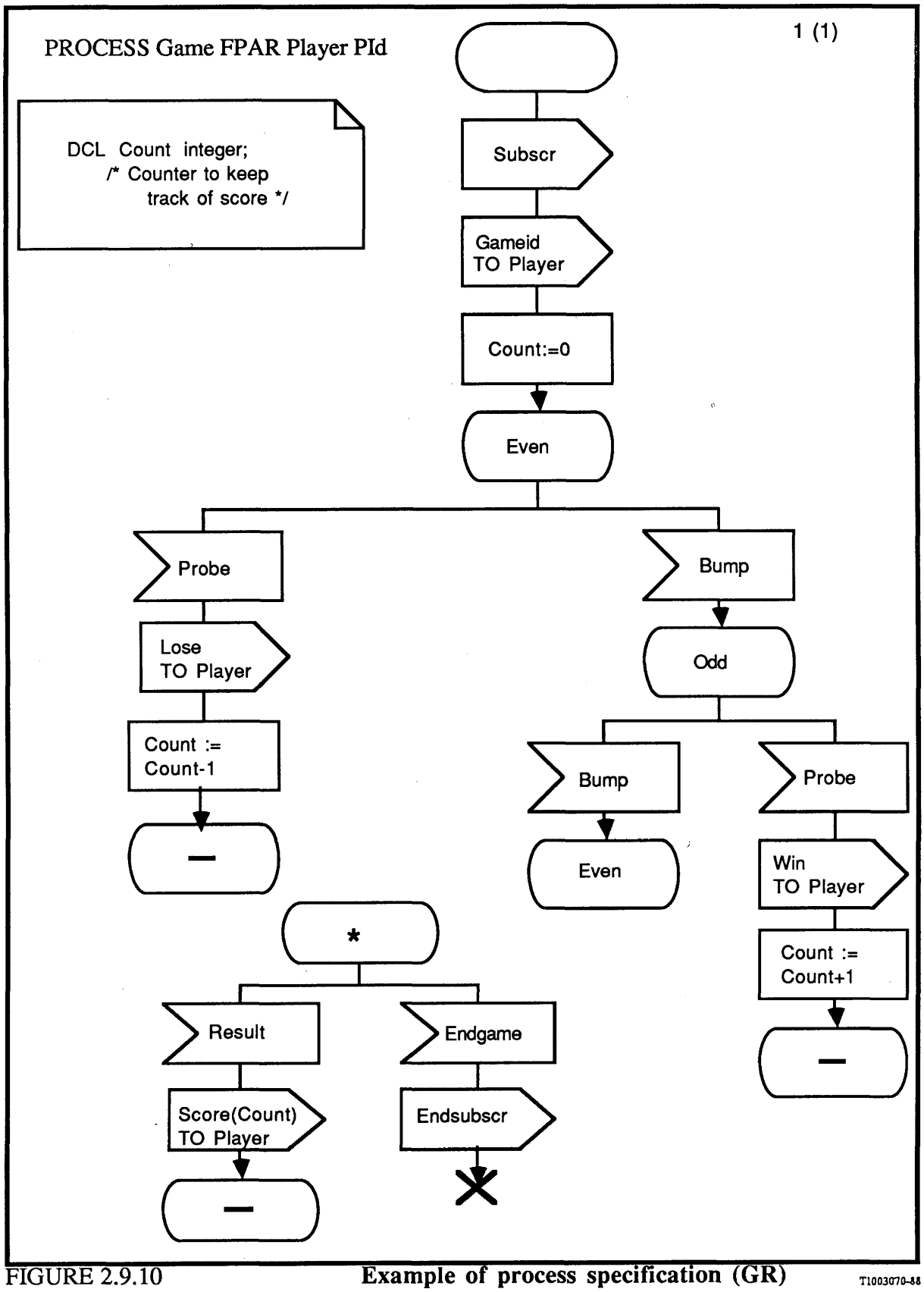


FIGURE 2.9.10 **Example of process specification (GR)**

```
PROCEDURE check;
/* The following signal definitions are assumed:
   SIGNAL sig1(Boolean), sig2, sig3(Integer,PId);   */
     FPAR IN/OUT x, y Integer;
     DCL  sum, index Integer,
          nice Boolean;
     START;
     TASK  sum := 0,
           index := 1;
     NEXTSTATE idle;
     STATE idle;
           INPUT sig1(nice);
           DECISION nice;
                 (true): TASK 'Calculate sum';
                       OUTPUT sig3(sum, SENDER);
                       RETURN;
                 (false): NEXTSTATE Jaj;
           ENDDECISION;
           INPUT sig2;
           ......
     .........
     .........

ENDPROCEDURE check;
```

FIGURE 2.9.11

Example of a fragment of a procedure specification (PR)

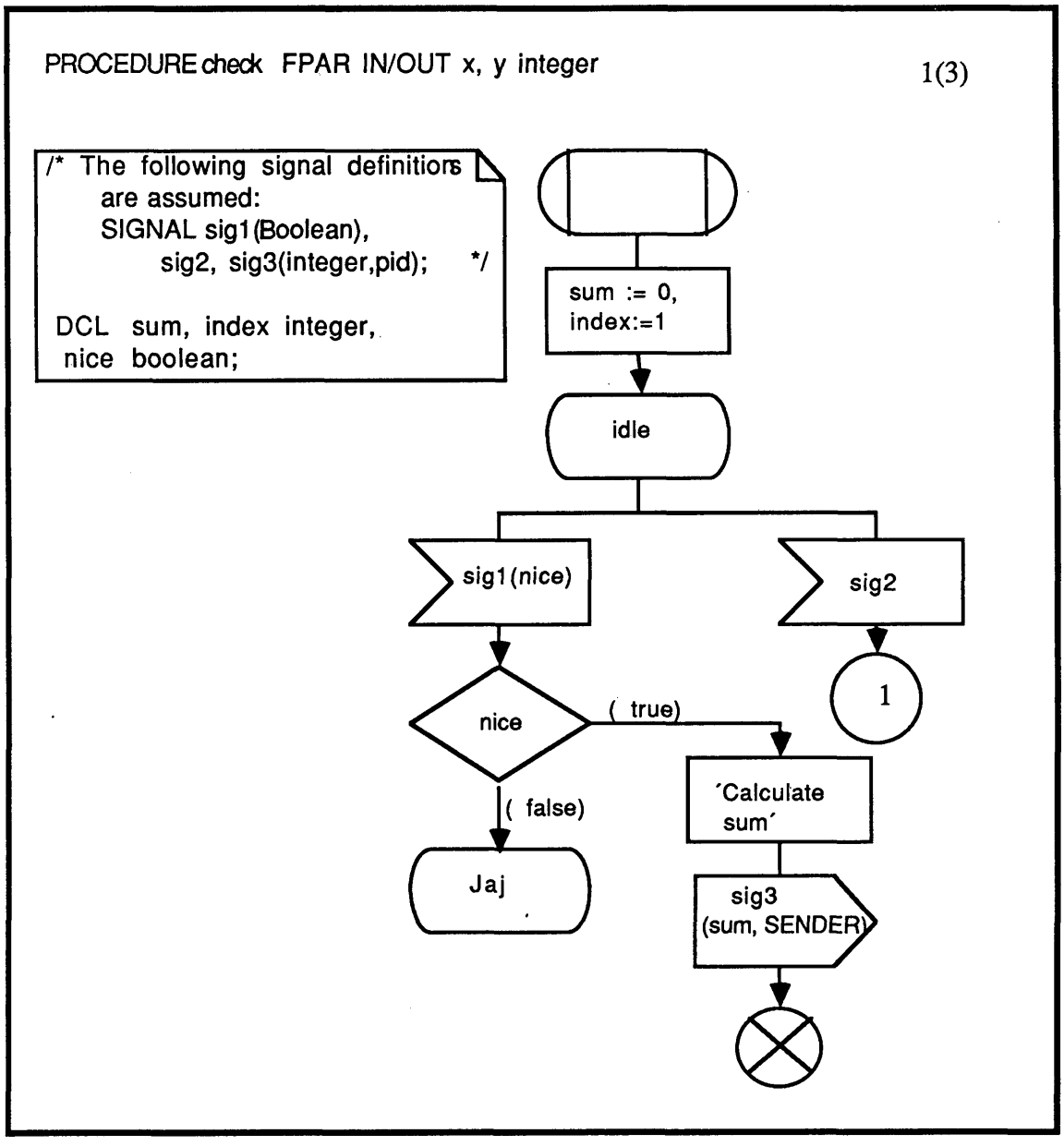


FIGURE 2.9.12

Example of a fragment of a procedure specification (GR)

```
/* The following signal definition is assumed:
SIGNAL inquire(Integer,Integer,Integer); */
PROCESS alfa;
DCL a,b,c Integer;
        .....
        ......
        INPUT inquire (a,b,c);
        CALL check (a,b);
        ......
ENDPROCESS;
```

FIGURE 2.9.13

Example of a procedure call in a fragment of a process definition (PR)

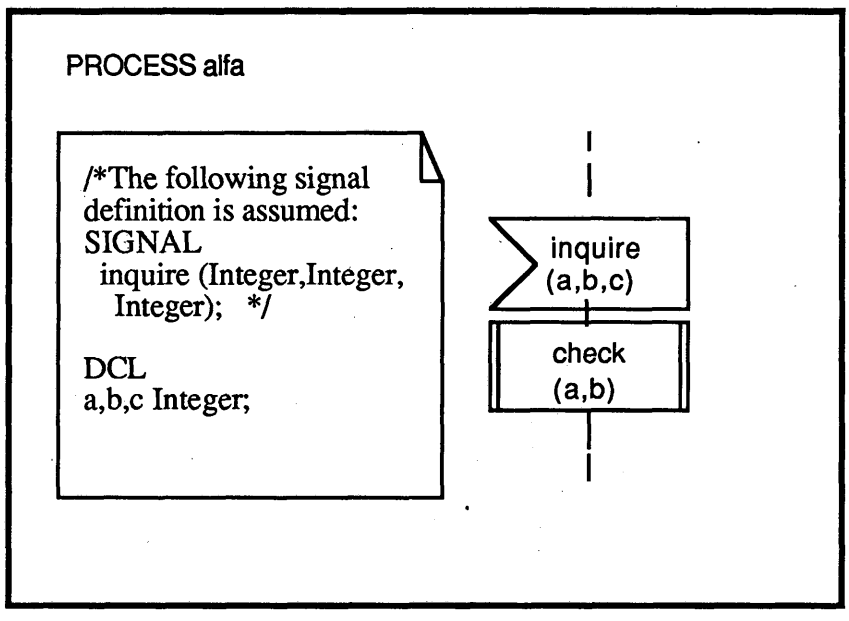


FIGURE 2.9.14

Example of a procedure call in a fragment of a process definition (GR)

3 Structural concepts in SDL

3.1 *Introduction*

This section defines a number of concepts needed to handle hierarchical structures in SDL. The basis for these concepts is defined in §2 and the defined concepts are strict additions to those defined in §2.

The intention with the concepts introduced in this section is to provide the user of SDL with means to specify large and/or complex systems. The concepts defined in §2 are suitable for specifying relatively small systems which may be understood and handled at a single level of blocks. When a larger, or complex system is specified, there is a need to partition the system specification into manageable units, which may be handled and understood independently. It is often suitable to perform the partitioning in a number of steps, resulting in a hierarchical structure of units specifying the system.

The term partitioning means subdivision of a unit into smaller subunits that are components of the unit. Partitioning does not affect the static interface of a unit. In addition to partitioning, there is also a need to add new details to the behaviour of a system when descending to lower levels in the hierarchical structure of the system specification. This is denoted by the term refinement.

3.2 *Partitioning*

3.2.1 *General*

A block definition may be partitioned into a set of subblock definitions, channel definitions and subchannel definitions. Similarly, a channel definition may be partitioned into a set of block definitions, channel definitions and subchannel definitions. Thus, each block definition and channel definition can have two versions: an unpartitioned version and a partitioned version in the concrete syntaxes. However channel substructures are transformed when mapping onto the abstract syntax. These two versions have the same static interface, but their behaviour may differ to some extent, because the order of output signals may not be the same. A subblock definition is a block definition, and a subchannel definition is a channel definition.

In a concrete system definition as well as in an abstract system definition, both the unpartitioned and the partitioned version of a block definition may appear. In such a case, a concrete system definition contains several consistent partitioning subsets, each subset corresponding to a system instance. A consistent partitioning subset is a selection of the *block definitions* in a *system definition* such that:

a) If it contains a *Block-definition*, then it must contain the definition of the enclosing scope unit if there is one;

b) It must contain all the *Block-definitions* defined on the system level and if it contains a *Sub-block-definitions* of a *Block-definition* , then it must also contain all other *Sub-block-definitions* of that *Block-definition*.

c) All "leaf" *Block-definitions* in the resulting structure contain *Process-definitions*.

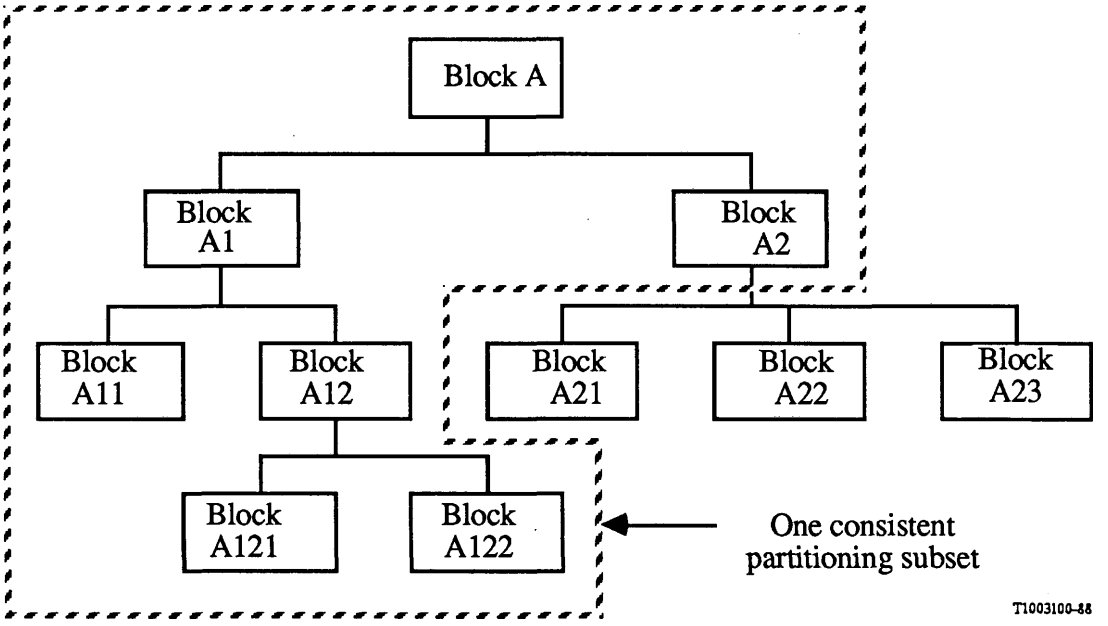


FIGURE §3.2.1

Consistent partitioning subset illustrated in an auxiliary diagram

At system interpretation time a consistent partitioning subset is interpreted.
The processes in each of the leaf blocks in the consistent partitioning subset are interpreted. If these leaf blocks also contain substructures, they have no effect. The substructures in the non-leaf blocks have an effect on visibility, and the processes in these blocks are not interpreted.

3.2.2 *Block partitioning*

Abstract grammar

Block-substructure-definition	::	*Block-substructure-name* *Sub-block-definition*-**set** *Channel-connection*-**set** *Channel-definition*-**set** *Signal-definition*-**set** *Data-type-definition* *Syn-type-definition*-**set**
Block-substructure-name	=	*Name*
Sub-block-definition	=	*Block-definition*
Channel-connection	::	*Channel-identifier* *Sub-channel-identifier*-**set**
Sub-channel-identifier	=	*Channel-identifier*
Channel-identifier	=	*Identifier*

The *Block-substructure-definition* must contain at least one *Sub-block-definition.* It is understood in the following that an abstract syntax term is contained in the *Block-substructure-definition* , if not stated otherwise.

A *Block-identifier* contained in a *Channel-definition* must denote a *Sub-block-definitions.* A *Channel-definition* connecting a *Sub-block-definition* to the boundary of the *Block-substructure-definition* is called a subchannel definition.

For each external *Channel-definition* connected to the *Block-substructure-definition* there must be exactly one *Channel-connection.* The *Channel-identifier* in the *Channel-connection* must identify this external *Channel-definition.*

For signals directed out of the *Block-substructure-definition,* the union of the *Signal-identifiers* associated to the *Sub-channel-identifier*-set contained in a *Channel-connection* must be identical to the *Signal-identifiers* associated to the *Channel-identifier* contained in the *Channel-connection.* The same rule is valid for signals directed into the *Block-substructure-definition.* However, this rule is modified in case of signal refinement, see §3.3.

Each *Sub-channel-identifier* must appear in one and only one *Channel-connection.*

Since a *Sub-block-definition* is a *Block-definition,* it may be partitioned. This partitioning may be repeated any number of times, resulting in a hierarchical tree structure of *Block-definitions* and their *Sub-block-definitions.* The *Sub-block-definitions* of a *Block-definition* are said to exist on the next lower level in the block tree, see also the figure below.

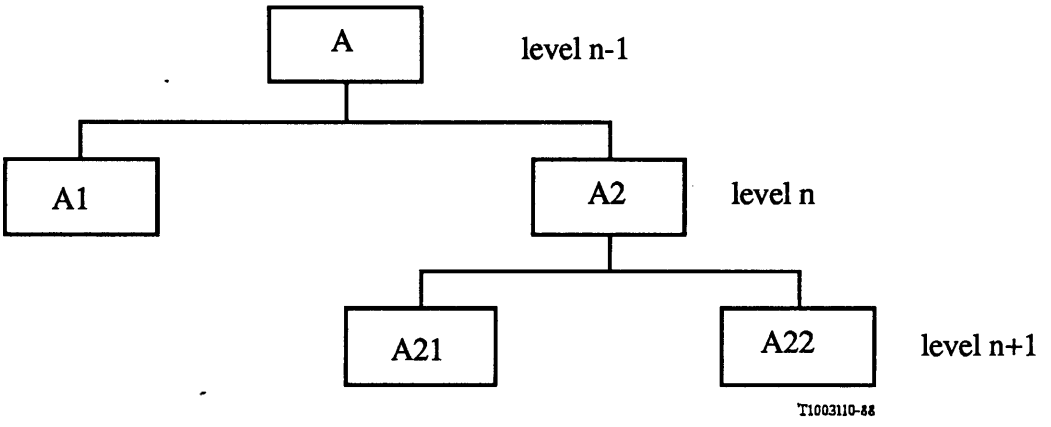


FIGURE 1/§3.2.2

A block tree diagram

The block tree diagram is an auxiliary diagram.

Concrete textual grammar

<block substructure definition> ::=
SUBSTRUCTURE {[<block substructure name>]
| <block substructure identifier> } <end>
{ <block definition>
| <textual block reference>
| <channel definition>
| <channel connection>
| <signal definition>
| <signal list definition>
| <data definition>
| <select definition>
| <macro definition> }+
ENDSUBSTRUCTURE [{ <block substructure name>
| <block substructure identifier>}] <end>

The <block substructure name> after the keyword SUBSTRUCTURE can be omitted only if it is the same as the <block name> in the enclosing <block definition>.

<textual block substructure reference> ::=
SUBSTRUCTURE <block substructure name> REFERENCED <end>

<channel connection> ::=
CONNECT <channel identifier> AND <subchannel identifier>
{, <subchannel identifier>}* <end>

Concrete graphical grammar

<block substructure diagram> ::=
<frame symbol>
contains {<block substructure heading>
{ {<block substructure text area>}*
{<macro diagram>}*
<block interaction area> }**set** }
is associated with {<channel identifier>}*

The <channel identifier> identifies a channel connected to a subchannel in the <block substructure diagram>. It is placed outside the <frame symbol> close to the endpoint of the subchannel at the <frame symbol>.

A <channel symbol> within the <frame symbol> and connected to it indicates a subchannel.

<block substructure heading> ::=
SUBSTRUCTURE {<block substructure name> | <block substructure identifier>}

<block substructure text area> ::=
<system text area>

<block substructure area> ::=
<graphical block substructure reference>
| <block substructure diagram>
| <open block substructure diagram>

<graphical block substructure reference> ::=
<block substructure symbol> **contains** <block substructure name>

<block substructure symbol> ::=
<block symbol>

< open block substructure diagram> ::=
{{<block substructure text area>}*
{<macro diagram>}*
<block interaction area>} **set**

When a <block substructure area> is an <open block substructure diagram>, then the enclosing <block diagram> must not contain <block text area>, <macro diagram> nor <process interaction area>.

Semantics

See § 3.2.1.

Model

An <open block substructure diagram> is transformed to a <block substructure diagram> in such a way that in the <block substructure heading> the <block substructure name> or the <block substructure identifier> is the same as the <block name> respectively <block identifier> in the enclosing <block diagram>.

Example

An example of a <block substructure definition> is given below.

```
BLOCK A;
    SUBSTRUCTURE A ;
        SIGNAL s5(nat), s6, s8, s9(min);
        BLOCK a1 REFERENCED;
        BLOCK a2  REFERENCED;
        BLOCK a3  REFERENCED;
        CHANNEL c1 FROM a2 TO ENV WITH s1, s2; ENDCHANNEL c1;
        CHANNEL c2 FROM ENV TO a1 WITH s3;
            FROM a1 TO ENV WITH s1;  ENDCHANNEL c2;
        CHANNEL d1 FROM a2 TO ENV WITH s7; ENDCHANNEL d1;
        CHANNEL d2 FROM a3 TO ENV WITH s10; ENDCHANNEL d2;
        CHANNEL e1 FROM a1 TO a2 WITH s5, s6; ENDCHANNEL e1;
        CHANNEL e2 FROM a3 TO a1 WITH s8; ENDCHANNEL e2;
        CHANNEL e3 FROM a2 TO a3 WITH s9; ENDCHANNEL e3;
        CONNECT c AND c1, c2 ;
        CONNECT d AND d1, d2 ;
    ENDSUBSTRUCTURE A;
ENDBLOCK A ;
```

The <block substructure diagram> for the same example is given below.

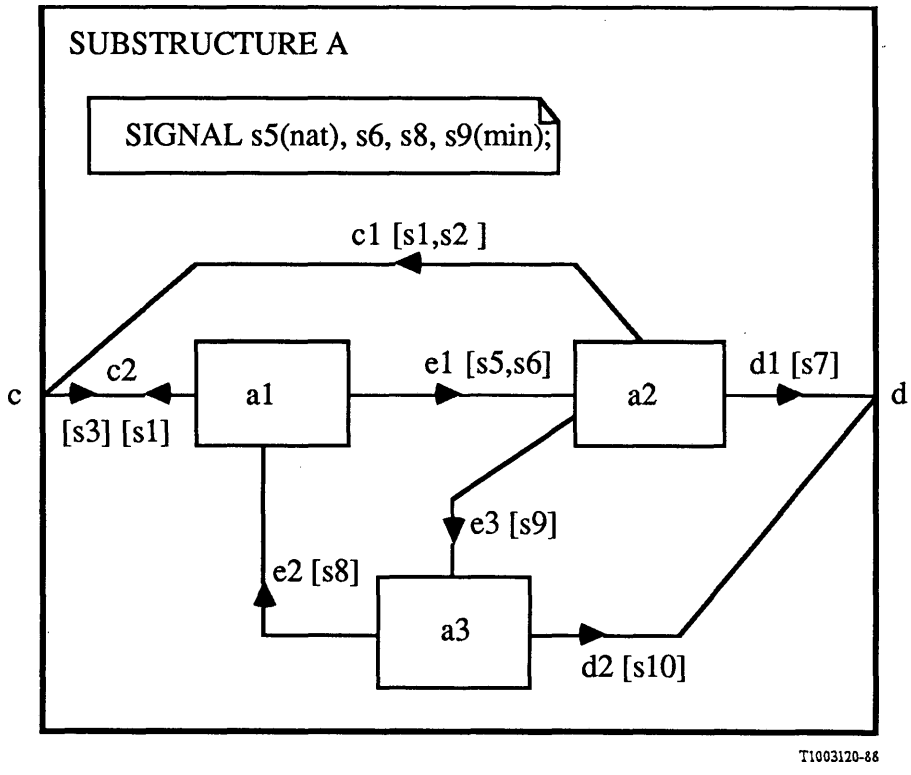


FIGURE 2/§3.2.2

Block substructure diagram for block A

3.2.3 *Channel partitioning*

All static conditions are stated using concrete textual grammar. Analogous conditions hold for the concrete graphical grammar.

Concrete textual grammar

<channel substructure definition> ::=
SUBSTRUCTURE {[<channel substructure name>]
| <channel substructure identifier> } <end>
{ <block definition>
| <textual block reference>
| <channel definition>
| <channel endpoint connection>
| <signal definition>
| <signal list definition>
| <data definition>
| <select definition>
| <macro definition> }+
ENDSUBSTRUCTURE [{ <channel substructure name>
| <channel substructure identifier>}] <end>

The <channel substructure name> after the keyword SUBSTRUCTURE can be omitted only if it is the same as the <channel name> in the enclosing <channel definition>.

<textual channel substructure reference> ::=
SUBSTRUCTURE <channel substructure name> REFERENCED <end>

<channel endpoint connection> ::=
CONNECT {<block identifier> | ENV} AND <subchannel identifier>
{, <subchannel identifier>}* <end>

For each endpoint of the partitioned <channel definition> there must be exactly one <channel endpoint connection>. The <block identifier> or ENVIRONMENT in a <channel endpoint connection> must identify one of the endpoints of the partitioned <channel definition>.

Concrete graphical grammar

<channel substructure diagram> ::=
<frame symbol>
contains {<channel substructure heading>
{ {<channel substructure text area>}*
{<macro diagram>}*
<block interaction area> }**set** }
is associated with {<block identifier> | ENV}+

The <block identifier> or ENV identifies an endpoint of the partitioned channel. The <block identifier> is placed outside the <frame symbol> close to the endpoint of the associated subchannel at the <frame symbol>. The <channel symbol> within the <frame symbol> and connected to this indicates a subchannel.

<channel substructure heading> ::=
SUBSTRUCTURE { < channel substructure name>
| < channel substructure identifier>}

<channel substructure text area> ::=
<system text area>

<channel substructure association area> ::=
<dashed association symbol>
is connected to <channel substructure area>

<channel substructure area> ::=
<graphical channel substructure reference>
| <channel substructure diagram>

<graphical channel substructure reference> ::=
<channel substructure symbol> **contains** <channel substructure name>

<channel substructure symbol> ::=
<block symbol>

Model

A <channel definition> which contains a <channel substructure definition> is transformed into a <block definition> and two <channel definition>s such that:

a) The two <channel definition>s are each connected to the block and to an endpoint of the original channel. The <channel definition>s have distinct new names and every reference to the original channel in the VIA constructs is replaced by a reference to the appropriate new channel.

b) The <block definition> has a distinct new name and it contains only a <block substructure definition> having the same name and containing the same definitions as the original <channel substructure definition>. The qualifiers in the new <block definition> are changed to include the block name. The two <channel endpoint connection>s from the original <channel substructure definition> are represented by two <channel connection>s wherein the <block identifier> or ENV is replaced by the appropriate new channel.

This transformation must take place immediately after those of a generic system. See § 4.3.

Example

An example of a <channel substructure definition> is given below.

```
CHANNEL C FROM A TO B  WITH s1;
      FROM B TO A WITH s2;

      SUBSTRUCTURE C;

            SIGNAL s3(hel), s4(boo), s5;

            BLOCK b1 REFERENCED;
            BLOCK b2 REFERENCED;

            CHANNEL c1 FROM ENV TO b1 WITH s1;
                  FROM b1 TO ENV WITH s2; ENDCHANNEL c1;

            CHANNEL c2 FROM b2 TO ENV WITH s1;
                  FROM ENV TO b2 WITH s2; ENDCHANNEL c2;

            CHANNEL e1 FROM b1 TO b2 WITH s3; ENDCHANNEL e1;
            CHANNEL e2 FROM b2 TO b1 WITH s4, s5; ENDCHANNEL e2;

            CONNECT A AND c1;
            CONNECT B AND c2;

      ENDSUBSTRUCTURE C;

ENDCHANNEL C;
```

The <channel substructure diagram> for the same example is given below.

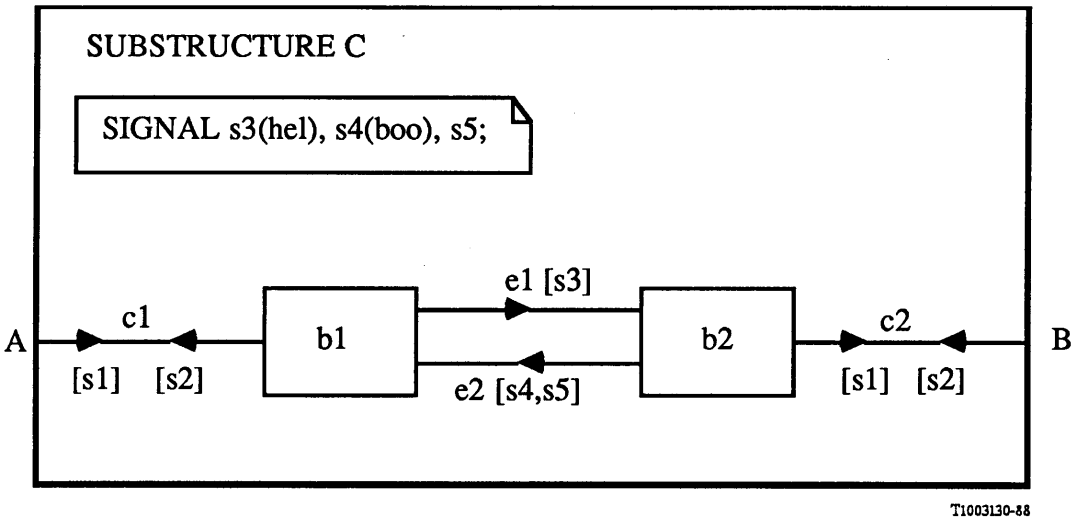


FIGURE §3.2.3

Channel substructure diagram for channel C

3.3 *Refinement*

Refinement is achieved by refining a signal definition into a set of subsignal definitions. A subsignal definition is a signal definition and may be refined. This refinement can be repeated any number of times, resulting in a hierarchical structure of signal definitions and their subsignal definitions. Note that a subsignal definition of a signal definition is not considered a component of the signal definition.

Abstract grammar

Signal-refinement :: *Subsignal-definition*-**set**

Subsignal-Definition :: [REVERSE] *Signal-definition*

For each *Channel-connection* it must hold that for each *Signal-identifier* associated to the *Channel-identifier* either the *Signal-identifier* is associated to at least one of the *Sub-channel-identifiers*, or each of its subsignal identifiers is associated to at least one of the *Sub-channel-identifiers*. This is a change of the corresponding rules for partitioning.

No two signals in the complete valid input signal set of a process definition or in the *Output-nodes* of a process definition may be on different refinement levels of the same signal.

Concrete textual grammar

<signal refinement> ::=
REFINEMENT
{<subsignal definition>}+
ENDREFINEMENT

<subsignal definition> ::=
[REVERSE] <signal definition>

Semantics

When a signal is defined to be carried by a channel, the channel will automatically be the carrier for all the subsignals of the signal. Refinement may take place when the channel is partitioned or split into subchannels. In such a case the subchannels will carry the subsignals in place of the refined signal. The direction of a subsignal is determined by the carrying subchannel, a subsignal may have an opposite direction to the refined signal, which is indicated by the keyword REVERSE. Signals cannot be refined when a channel is split into signal routes.

When a system definition contains signal refinement, the concept of consistent partitioning subset is restricted. Such a system definition is said to contain several consistent refinement subsets.

A consistent refinement subset is a consistent partitioning subset restricted by the following rule:

- When selecting the consistent partitioning subset, the set of signals on signalroutes connected to an endpoint of a channel must not contain parent signals of contained subsignals, and unless the other endpoint is the system ENVIRONMENT, the set of signals for the first endpoint must be equal to the set of signals on signalroutes connected to the other endpoint.

Example

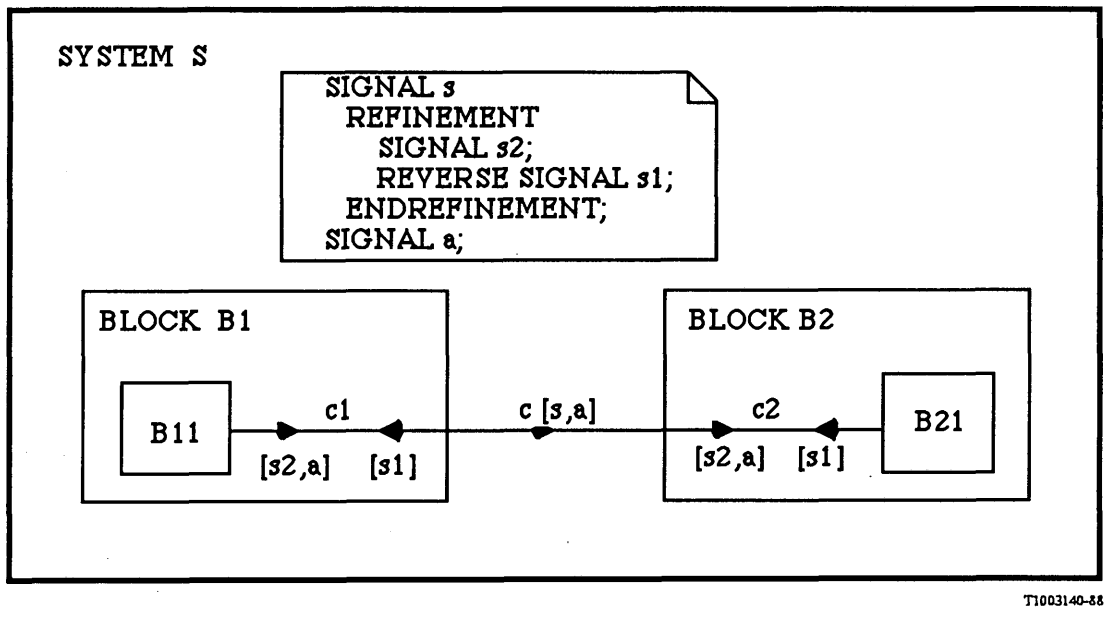


FIGURE §3.3

System diagram containing signal refinement

In the above example signal s is refined in block definition B1 and B2, but not signal a. On the highest refinement level, processes in B1 and B2 are communicating using signal s and a. On the next lower level, processes in B11 and B21 are communicating using s1, s2 and a.

Note that refinement in only one of the block definitions B1 and B2 is not allowed, since there is no dynamic transformation between a signal and its subsignals, only a static relation.

4 Additional concepts in SDL

4. 1 *Introduction*

This chapter defines a number of additional concepts. These additional concepts are standard shorthand notations, and are modeled in terms of the primitive concepts of SDL, using concrete syntax. They are introduced for the convenience of the users of SDL in addition to shorthand notations defined in other chapters of the Recommendation.

The properties of a shorthand notation is derived from the way it is modeled in terms of (or transformed to) the primitive concepts. In order to ensure easy and unambiguous use of the shorthand notations, and to reduce side effects when several shorthand notations are combined, these concepts are transformed in a specified order as follows:

1 Macro § 4.2
2 Generic systems § 4.3
3 Asterisk state § 4.4
4 State list § 2.6.3
5 Multiple appearence of state § 4.5
6 Asterisk input § 4.6
7 Asterisk save § 4.7
8 Stimulus list § 2.6.4
9 Output list § 2.7.4
10 Implicit transition § 4.8
11 Dash nextstate § 4.9
12 Service § 4.10
13 Continuous signal § 4.11
14 Enabling condition § 4.12
15 Imported and exported value § 4.13

This order is also followed when defining the concepts in this section. The specified order of transformation means that in the transformation of a shorthand notation of order *n* , another shorthand notation of order *m* may be used, provided m>n.

Since there is no abstract syntax for the shorthand notations, terms of either graphical syntax or textual syntax are used in their definitions. The choice between graphical syntax terms and textual syntax terms is based on practical considerations, and does not restrict the use of the shorthand notations to a particular concrete syntax.

4.2 *Macro*

In the following text the terms macro definition and macro call are used in a general sense, covering both SDL/GR and SDL/PR. A macro definition contains a collection of graphical symbols and/or lexical units, that can be included in one or more places in the <concrete system definition>. Each such place is indicated by a macro call. Before a <concrete system definition> can be analysed, each macro call must be replaced by the corresponding macro definition.

4.2.1 *Lexical rules*

<formal name> ::=
[<name>%] <macro parameter>
{% <name> %<macro parameter> | %<macro parameter> }* [%<name>]

4.2.2 *Macro definition*

Concrete textual grammar

<macro definition> ::=
MACRODEFINITION <macro name>
[< macro formal parameters>] <end>
<macro body>
ENDMACRO [<macro name>] <end>

<macro formal parameters> ::=
FPAR < macro formal parameter> {, < macro formal parameter>}*

<macro formal parameter> ::=
<name>

<macro body> ::=
{<lexical unit>|<formal name>}*

<macro parameter> ::=
<macro formal parameter>
| MACROID

The <macro formal parameter>s must be distinct. <macro actual parameter>s of a macro call must be matched one to one with their corresponding <macro formal parameter>s.

The <macro body> must not contain the keyword ENDMACRO and MACRODEFINITION.

Concrete graphical grammar

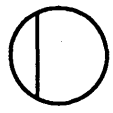
<macro diagram> ::=
<frame symbol> **contains** {<macro heading> <macro body area>}

<macro heading> ::=
MACRODEFINITION <macro name> [<macro formal parameters>]

<macro body area> ::=
{ {<any area> }*
<any area> [**is connected to** <macro body port1>] }**set**
|{ <any area> **is connected to** <macro body port2>
<any area> **is connected to** <macro body port2>
{ <any area> [**is connected to** <macro body port2>]}*}**set**

<macro inlet symbol>::=

<macro outlet symbol>::=



<macro body port1> ::=
<outlet symbol> **is connected to** {<frame symbol>
[is associated with <macro label>]
| <macro inlet symbol> [{**contains** |**is associated with** } <macro label>]
| <macro outlet symbol> [{**contains** |**is associated with** } <macro label>] }

<macro body port2> ::=
<outlet symbol> **is connected to** {<frame symbol>
is associated with <macro label>
| <macro inlet symbol> {**contains** |**is associated with** } <macro label>
| <macro outlet symbol> {**contains** |**is associated with** } <macro label>}

<macro label> ::=
<name>

<outlet symbol> ::=
<dummy outlet symbol>
| <flow line symbol>
| <channel symbol>
| <signal route symbol>
| <solid association symbol>
| <dashed association symbol>
| <create line symbol>

<dummy outlet symbol> ::=
<solid association symbol>

<any area> ::=
<system text area>
| <block interaction area>
| <signal list area>
| <block area>
| <block text area>
| <process interaction area>
| <graphical procedure reference>
| <process text area>
| <process graph area>
| <merge area>
| <transition string area>
| <state area>
| <input area>
| <save area>
| <set area>
| <reset area>
| <export area>
| <text extension area>
| <channel substructure association area>
| <channel substructure area>

| <block substructure area>
| <priority input area>
| <continuous signal area>
| <in-connector area>
| <nextstate area>
| <process area>
| <channel definition area>
| <create line area>
| <signal route definition area>
| <graphical process reference>
| <process diagram>
| <start area>
| <output area>
| <priority output area>
| <task area>
| <create request area>
| <procedure call area>
| <procedure area>
| <decision area>
| <out-connector area>
| <procedure text area>
| <procedure graph area>
| <procedure start area>
| <block substructure text area>
| <block interaction area>
| <service area>
| <service signal route definition area>
| <service text area>
| <service graph area>
| <service start area>
| <comment area>
| <macro call area>
| <input association area>
| <save association area>
| <option area>
| <channel substructure text area>
| <transition option area>
| <service interaction area>
| <priority input association area>
| <contionuous signal association area>
| <enabling condition area>

A <dummy outlet symbol> must not have anything associated to it except for <macro label>.

For an <outlet symbol> which is not a <dummy outlet symbol>, the corresponding <inlet symbol> in the macro call must be a <dummy inlet symbol>.

A <macro body> may appear in any text referred to in <any area>.

Semantics

A <macro definition> contains lexical units, while a <macro diagram> contains syntactical units. Thus, mapping between macro constructs in textual syntax and graphical syntax is generally not possible. For the same reason, separate detailed rules apply for textual syntax and graphical syntax, although there are some common rules.

<macro name> is visible in the whole system definition, no matter where the macro definition appears. A macro call may appear before the corresponding macro definition.

A macro definition may contain macro calls, but a macro definition must not call itself either directly or indirectly through macro calls in other macro definitions.

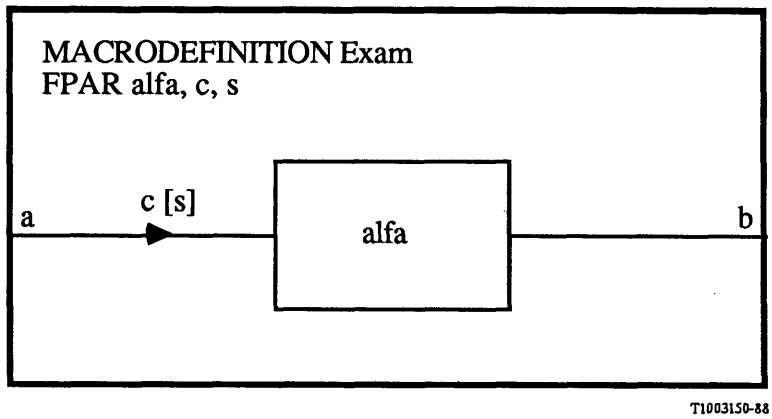
The keyword MACROID may be used as a pseudo macro formal parameter within each macro definition. No <macro actual parameter>s can be given to it, and it is replaced by a unique <name> for each expansion of a macro definition (within an expansion the same <name> is used for each occurence of MACROID).

Example

Below is given an example of a <macro definition>:

```
MACRODEFINITION Exam
FPAR alfa, c, s, x;
     BLOCK alfa REFERENCED;
     CHANNEL c  FROM x TO alfa WITH s; ENDCHANNEL c;
ENDMACRO Exam;
```

The <macro diagram> for the same example is given below. However, the <macro formal parameter>, x, is not required in this case.



T1003150-88

4.2.3 *Macro call*

Concrete textual grammar

<macro call> ::=
MACRO <macro name> [<macro call body>] <end>

<macro call body> ::=
(<macro actual parameter> {, <macro actual parameter>}*)

<macro actual parameter> ::=
{<lexical unit>}*

The <lexical unit> cannot be a comma "," or right parenthesis ")". If any of these two characters is required in a <macro actual parameter>, then the <macro actual parameter> must be a <character string>. If the <macro actual parameter> is a <character string>, then the value of the <character string> is used when the <macro actual parameter> replaces a <macro formal parameter>.

A <macro call> may appear at any place where a <lexical unit> is allowed.

Concrete graphical grammar

<macro call area> ::=
<macro call symbol> **contains** {<macro name> [<macro call body>]}
[**is connected to**
{<macro call port1> | <macro call port2> {<macro call port2>}+}]

<macro call symbol> ::=

<macro call port1> ::=
<inlet symbol> [**is associated with** <macro label>]
is connected to <any area>

<macro call port2> ::=
<inlet symbol> **is associated with** <macro label>
is connected to <any area>

<inlet symbol> ::=
<dummy inlet symbol>
| <flow line symbol>
| <channel symbol>.
| <signal route symbol>
| <solid association symbol>
| <dashed association symbol>
| <create line symbol>

<dummy inlet symbol> ::=
<solid association symbol>

A <dummy inlet symbol> must not have anything associated to it except for <macro label>.

For each <inlet symbol> there must be an <outlet symbol> in the corresponding <macro diagram>, associated with the same <macro label>. For an <inlet symbol> which is not a <dummy inlet symbol>, the corresponding <outlet symbol> must be a <dummy outlet symbol>.

Except in the case of <dummy inlet symbol>s and <dummy outlet symbol>s, it is possible to have multiple (textual) <lexical unit>s associated with an <inlet symbol> or <outlet symbol>. In this case the <lexical unit> closest to the <macro call symbol> or the <frame symbol> of the <macro diagram> is taken to be the <macro label> associated with the <inlet symbol> or <outlet symbol>.

The <macro call area> may appear at any place where an area is allowed. However, a certain space is required between the <macro call symbol> and any other closed graphical symbol. If such a space must not be empty according to the syntax rules, then the <macro call symbol> is connected to the closed graphical symbol with a <dummy inlet symbol>.

Semantics

A system definition may contain macro definitions and macro calls. Before such a system definition can be analysed, all macro calls must be expanded. The expansion of a macro call means that a copy of the macro definition having the same <macro name> as that given in the macro call replaces the macro call.

When a macro definition is called, it is expanded. This means that a copy of the macro definition is created, and each occurence of the <macro formal parameter>s of the copy is replaced by the corresponding <macro actual parameter>s of the macro call, then macro calls in the copy, if any, are expanded. All percent characters (%) in <formal name>s are removed when <macro formal parameter>s are replaced by <macro actual parameter>s.

There should be one to one correspondence between <macro formal parameter> and <macro actual parameter>.

- *Rules for graphical syntax*

  The <macro call area> is replaced by a copy of the <macro diagram> in the following way. All <macro inlet symbol>s and <macro outlet symbol>s are deleted. A <dummy outlet symbol> is replaced by the <inlet symbol> having the same <macro label>. A <dummy inlet symbol> is replaced by the <outlet symbol> having the same <macro label>. Then the <macro label>s attached to <inlet symbol>s and <outlet symbol>s are deleted. <macro body port1> and <macro boby port2> which have no corresponding <macro call port1> or <macro call port2> are also deleted.

Example

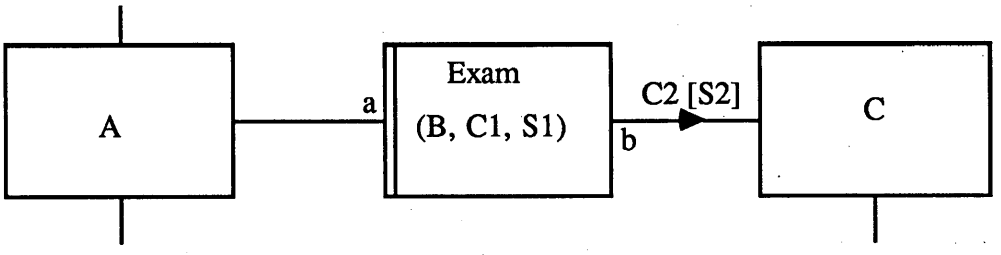
Below is given an example of a <macro call>, within a fragment of a <block definition>.

```
.....
BLOCK A REFERENCED;
MACRO Exam (B, C1, S1, A);
BLOCK C REFERENCED;
CHANNEL C2 FROM B TO C WITH S2; ENDCHANNEL C2;
.....
```

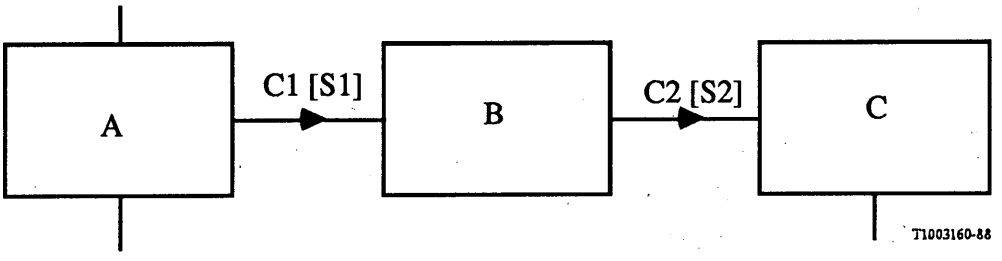
The expansion of this macro call, using the example in §4.2.2, gives the following result.

```
.....
BLOCK A REFERENCED;
BLOCK B REFERENCED;
CHANNEL C1 FROM A TO B WITH S1; ENDCHANNEL C1;
BLOCK C REFERENCED;
CHANNEL C2 FROM B TO C WITH S2; ENDCHANNEL C2;
.....
```

The <macro call area> for the same example, within a fragment of a <block interaction area>, is given below.



The expansion of this macro call gives the following result.



4.3 *Generic systems*

A system specification may have optional parts and system parameters with undefined values in order to meet various needs. Such a system specification is called generic, its generic property is specified by means of external synonyms (which are analogous to formal parameters in a procedure definition). A generic system specification is tailored by selecting a suitable subset of it and providing a value for each of the system parameters. The resulting system specification does not contain external synonyms, and is called a specific system specification.

4.3.1 *External synonym*

Concrete textual grammar

<external synonym definition>::=
SYNONYM <external synonym name> <predefined sort> = EXTERNAL

<external synonym> ::=
<external synonym identifier>

An <external synonym definition> may appear at any place where a <synonym definition> is allowed, see §5.4.1.13. An <external synonym> may be used at any place where a <synonym> is allowed, see §5.4.2.3. The predefined sorts are: Boolean, Character, Charstring, Integer, Natural, Real, PId, Duration or Time.

Semantics

An <external synonym> is a <synonym> whose value is not specified in the system definition. This is indicated by the keyword EXTERNAL which is used instead of a <simple expression>.

A generic system definition is a system definition that contains <external synonym>s, or <informal text> in a transition option (see §4.3.4). A specific system definition is created from a generic system definition by providing values for the <external synonym>s, and transforming <informal text> to formal constructs. How this is accomplished, and the relation to the abstract grammar, is not part of the language definition.

4.3.2 *Simple expression*

Concrete textual grammar

<simple expression> ::=
<ground expression>

A <simple expression> must only contain operators, synonyms and literals of the predefined sorts.

Semantics

A simple expression is a *Ground-expression.*

4.3.3 *Optional definition*

Concrete textual grammar

<select definition> ::=
SELECT IF (<boolean simple expression>) <end>
{<block definition>
| <textual block reference>
| <channel definition>
| <signal definition>
| <signal list definition>
| <data definition>
| <process definition>
| <textual process reference>
| <timer definition>
| <service signal route definition>
| <channel connection>
| <channel endpoint connection>
| <variable definition>
| <view definition>
| <import definition>
| <procedure definition>
| <textual procedure reference>
| <service definition>
| <textual service reference>
| <signal route definition>
| <channel to route connection>
| <signal route connection>
| <select definition>
| <macro definition> }+
ENDSELECT <end>

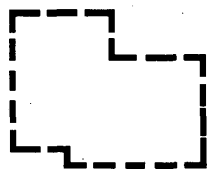
The <boolean simple expression> must not be dependent on any definition within the <select definition>. A <select definition> must contain only those definitions that are syntactically allowed at that place.

Concrete graphical grammar

<option area> ::=
<option symbol> **contains**
{ SELECT IF (<boolean simple expression>)
{<block area>
| <channel definition area>
| <system text area>
| <block text area>
| <process text area>
| <procedure text area>
| <block substructure text area>
| <channel substructure text area>
| <service text area>
| <macro diagram>

| <process area>
| <signal route definition area>
| <create line area>
| <procedure area>
| <option area>
| <service area>
| <service signal route definition area> }+ }

The <option symbol> is a dashed polygon having solid corners, for example:



An <option symbol> logically contains the whole of any one-dimensional graphical symbol cut by its boundary (i.e. with one end point outside).

The <boolean simple expression> must not be dependent on any area or diagram within the <option area>.

An <option area> may appear anywhere, except within a <process graph area>, <procedure graph area> and <service graph area>. An <option area> must contain only those areas and diagrams that are syntactically allowed at that place.

Semantics

If the value of the <boolean simple expression> is false, then the constructs contained in the <select definition> and <option symbol> are not selected. In the other case the constructs are selected.

Model

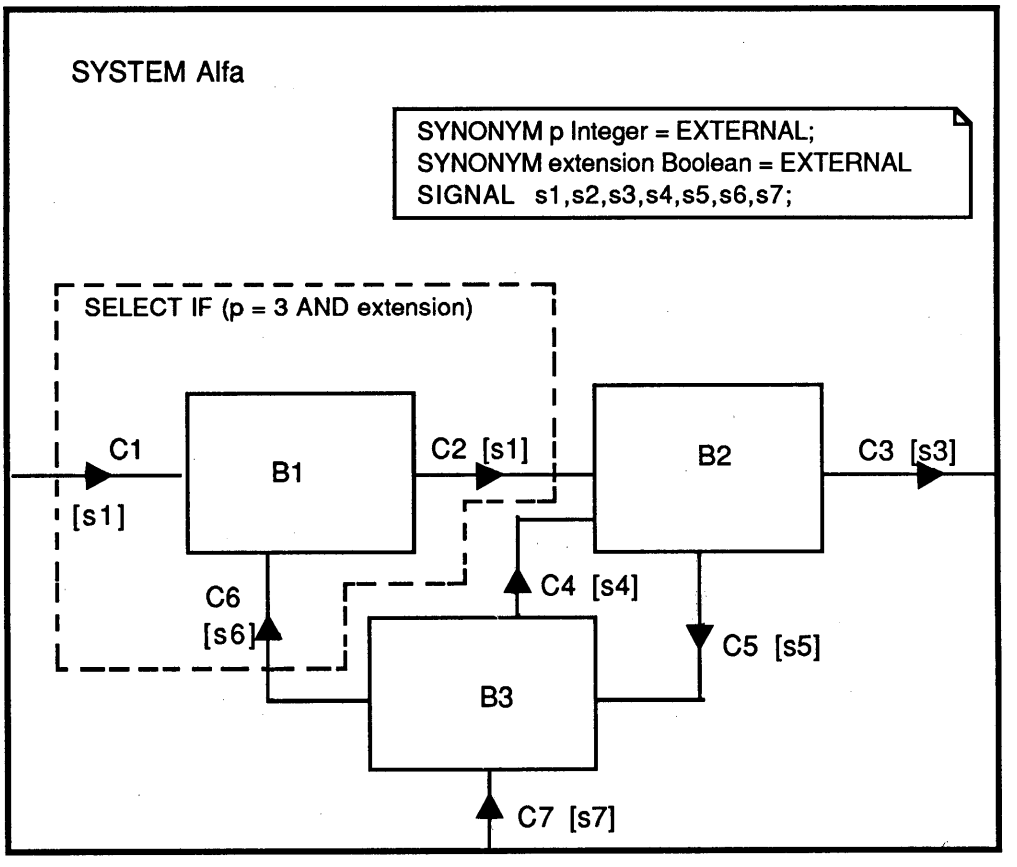
The <select definition> and the <option area> are deleted at transformation and are replaced by the contained selected constructs, if any.

Example

In system Alfa there are three blocks: B1, B2 and B3. Block B1 and the channels connected to it are optional, dependent on the values of the external synonyms p and extension. In SDL/PR this example is represented as follows.

```
SYSTEM Alfa;
      SYNONYM p Integer = EXTERNAL;
      SYNONYM extension Boolean = EXTERNAL;
      SIGNAL  s1,s2,s3,s4,s5,s6,s7;
      SELECT  IF (p = 3 AND extension);
        BLOCK B1 REFERENCED;
        CHANNEL C1 FROM ENV TO B1 WITH s1 ; ENDCHANNEL C1;
        CHANNEL C2 FROM B1 TO B2 WITH s2 ; ENDCHANNEL C2;
        CHANNEL C6 FROM B3 TO B1 WITH s6; ENDCHANNEL C6;
      ENDSELECT;
      CHANNEL C3 FROM B2 TO ENV WITH s3 ; ENDCHANNEL C3;
      CHANNEL C4 FROM B3 TO B2 WITH s4 ; ENDCHANNEL C4;
      CHANNEL C5 FROM B2 TO B3 WITH s5; ENDCHANNEL C5;
      CHANNEL C7 FROM ENV TO B3 WITH s7 ; ENDCHANNEL C7;
      BLOCK B2 REFERENCED;
      BLOCK B3 REFERENCED;
ENDSYSTEM Alfa;
```

The same example is in SDL/GR syntax represented as shown below.



4.3.4 *Optional transition string*

Concrete textual grammar

<transition option> ::=
ALTERNATIVE <alternative question> <end>
{<answer part> <else part>
| <answer part> { <answer part> }+ [<else part>] }
ENDALTERNATIVE

<alternative question>::=
<simple expression>
| <informal text>

Every <ground expression> in <answer> must be a <simple expression>. The <answer>s in a <transition option> must be mutually exclusive. If the <alternative question> is an <expression>, the *Range-condition* of the <answer>s must be of the same sort as of the <alternative question>.

Concrete graphical grammar

<transition option area> ::=
<transition option symbol> **contains** {<alternative question>}
is followed by {<option outlet1> {<option outlet1> | <option outlet2> }
{ <option outlet1> }* }**set**

<transition option symbol> ::=

<option outlet1> ::=
<flow line symbol> **is associated with** <graphical answer>
is followed by <transition area>

<option outlet2> ::=
<flow line symbol> **is associated with** ELSE
is followed by <transition area>

The <flow line symbol> in <option outlet1> and <option outlet2> is connected to the bottom of the <transition option symbol>. The <flow line symbol>s originating from a <transition option symbol> may have a common originating path. The <graphical answer> and ELSE may be placed along the associated <flow line symbol>, or in the broken <flow line symbol>.

The <graphical answer>s in a <transition option area> must be mutually exclusive.

Semantics

Constructs in an <option outlet1> are selected if the <answer> contains the value of the <alternative question>. If none of the <answer>s contains the value of the <alternative question>, then the constructs in the <option outlet2> are selected.

If no <option outlet2> is provided and none of the outgoing paths is selected then the selection is invalid.

Model

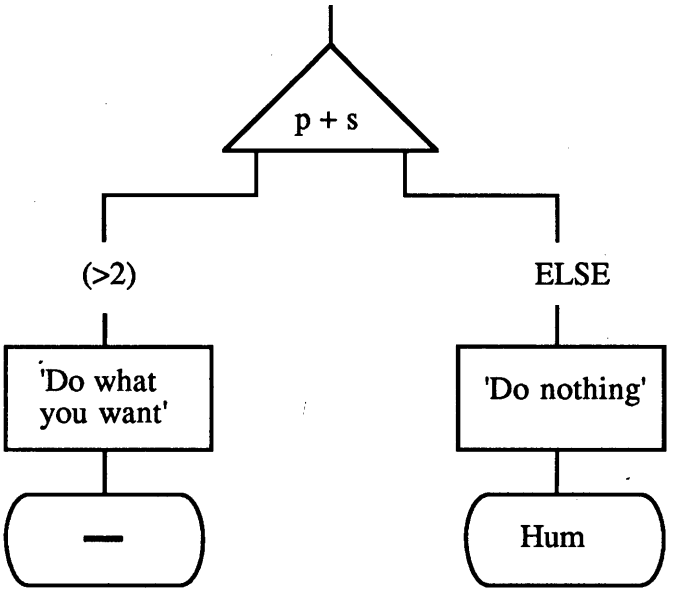
The <transition option> and <transition option area> is deleted at transformation and is replaced by the contained selected constructs.

Example

A fragment of a <process definition> containing a <transition option> is shown below. *p* and *s* are synonyms.

```
.....
ALTERNATIVE p + s;
        (>2) : TASK 'Do what you want';
               NEXTSTATE -;
        ELSE: TASK 'Do nothing';
               NEXTSTATE Hum;
ENDALTERNATIVE;
.....
```

The same example in concrete graphical syntax is shown below.



4.4 *Asterisk state*

Concrete textual grammar

<asterisk state list> ::=
<asterisk> [(<state name> {, <state name>}*)]

<asterisk> ::=
*

In a <process body>, <procedure body> or <service body>, at least one <state list> must be different from <asterisk state list>. The <state name>s in an <asterisk state list> must be distinct and must be contained in other <state list>s in the enclosing <process body>, <procedure body> or <service body>.

The <state name>s in the <asterisk state list> must not include all <state name>s in the enclosing <process body>, <procedure body> or <service body>.

Concrete graphical grammar

A <state area> containing <asterisk state list> must not coincide with a <nextstate area>.

Model

An <asterisk state list> is transformed to a <state list> containing all <state name>s of the <process body>, <service body> or <procedure body> in question, except for those <state name>s contained in the <asterisk state list>.

4.5 *Multiple appearance of state*

Concrete textual grammar

A <state name> may appear in more than one <state> of a <process body>, <service body> or <procedure body>.

Model

When several <state>s contain the same <state name>, these <state>s are concatenated into one <state> having that <state name>.

4.6 *Asterisk input*

Concrete textual grammar

<asterisk input list> ::=
<asterisk>

A <state> may contain at most one <asterisk input list>. A <state> must not contain both <asterisk input list> and <asterisk save list>.

Model

An <asterisk input list> is transformed to a list of <stimulus>s containing the complete valid input signal set of the enclosing <process definition> or <service definition>, except for <signal identifier>s of implicit signals and for <signal identifier>s contained in the other <input list>s and <save list>s of the <state>, and in all <priority input>s of the <service definition> § 4.10.

4.7 *Asterisk save*

Concrete textual grammar

<asterisk save list> ::=
<asterisk>

A <state> may contain at most one <asterisk save list>. A <state> must not contain both <asterisk input list> and <asterisk save list>.

Model

An <asterisk save list> is transformed to a list of <stimulus>s containing the complete valid input signal set of the enclosing <process definition> or <service definition>, except for <signal identifier>s of implicit signals and for <signal identifier>s contained in the other <input list>s and <save list>s of the <state>, and in all <priority input>s of the <service definition> § 4.10.

4.8 *Implicit transition*

Concrete textual grammar

A <signal identifier> contained in the complete valid input signal set of a <process definition> or <service definition> may be omitted in the set of <signal identifier>s contained in the <input list>s, <priority input list>s and the <save list> of a <state>.

Model

For each <state> there is an implicit <input part> containing a <transition> which only contains a <nextstate> leading back to the same <state>.

4.9 *Dash nextstate*

Concrete textual grammar

<dash nextstate> ::=
<hyphen>

<hyphen>::=
-

The <transition> contained in a <start> must not lead, directly or indirectly, to a <dash nextstate>.

Model

In each <nextstate> of a <state> the <dash nextstate> is replaced by the <state name> of the <state>.

4.10 *Service*

The behaviour of a process in basic SDL is defined by a process graph. The service concept offers an alternative to the process graph through a set of service definitions. In many situations service definitions can reduce the overall complexity and increase the readability of a process definition. In addition, each service definition may define a partial behaviour of the process, which may be useful in some applications.

4.10.1 *Service decomposition*

Concrete textual grammar

<service decomposition> ::=
{<service signal route definition>
| <signal route connection>
| <service definition>
| <select definition>
| <textual service reference>}+

<service signal route definition> ::=
SIGNALROUTE <service signal route name>
<service signal route path>
[<service signal route path>]

<service signal route path> ::=
{FROM <service identifier> TO <service identifier>
| FROM <service identifier> TO ENV
| FROM ENV TO <service identifier> }
WITH <signal list> <end>

<signal route connection> ::=
CONNECT <signal route identifier>
AND <service signal route identifier> {, <service signal route identifier>}* <end>

<textual service reference> ::=
SERVICE <service name> REFERENCED <end>

When a <process definition> contains a <service decomposition>, it must not contain <timer definition>s outside the <service decomposition>.

A <service decomposition> must contain at least one <service definition>.

Similar wellformedness rules apply for <service signal route> as for <signal route>.

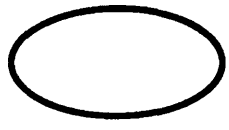
Concrete graphical grammar

<service interaction area> ::=
{ <service area> | <service signal route definition area> }+

<service area> ::=
<graphical service reference>
| <service diagram>

<graphical service reference> ::=
<service symbol> **contains** <service name>

<service symbol> ::=



<service signal route definition area> ::=
<signal route symbol>
is associated with {<service signal route name>
[<signal route identifier>]
<signal list area>
[<signal list area>] }**set**
is connected to {<service area>
{<service area> | <frame symbol>} }**set**

When the <signal route symbol> is connected to the <frame symbol>, then the <signal route identifier> identifies an external signal route to which the signal route is connected.

Semantics

The <service decomposition> is an alternative to the <process body>, and expresses the same behaviour.

Model

The service concept is modeled by transforming the <service decomposition> to primitive concepts. Transformation of <service signal route definition>s and <signal route connection>s results in nothing.

4.10.2 *Service definition*

Concrete textual grammar

```
<service definition> ::=
        SERVICE {<service name> | <service identifier>} <end>
        [<valid input signal set>]
        {<variable definition>
            | <data definition>
            | <timer definition>
            | <view definition>
            | <import definition>
            | <select definition>
            | <macro definition>
            | <procedure definition>
            | <textual procedure reference>}*
        <service body>
        ENDSERVICE [{<service name> | <service identifier>}] <end>

<service body> ::=
        <process body>

<priority input> ::=
        PRIORITY INPUT <priority input list> <end> <transition>

<priority input list> ::=
        <priority  stimulus> {, <priority stimulus>}*

<priority stimulus>  ::=
        <priority signal identifier> [ ( [<variable identifier>] {, [<variable identifier>] }* ) ]

<priority output> ::=
        PRIORITY OUTPUT <priority output body>

<priority output body> ::=
        <priority signal identifier> [<actual parameters>]
        {, <priority signal identifier> [<actual parameters>]}*
```

A signal is a high priority signal in a process if and only if it is mentioned in a <priority input> of a <service definition> in that process.

A <variable definition> in a <service definition> must not contain the keyword EXPORTED or REVEALED.

A <priority signal identifier> in a <priority output> must not be contained in an <input part> or in a <save part>. A <priority signal identifier> in a <priority input> must not be contained in an <output>.

The same rule on valid input signal set and service signal route stated in 2.5.2 on process applies.

The <service decomposition> may contain <service signal route definition>s only if the enclosing <block definition> contains <signal route definition>s.

Only one of the <service definition>s in a <service decomposition> is allowed to have a <start> containing a <transition string>. All other <start>s must contain only <nextstate>.

The complete valid input signal sets (each such sets being a union of the <valid input signal set> and the set of signals conveyed on incoming <service signal route>s of a <service definition>) of the <service definition>s within a <process definition> must be disjoint.

A <procedure definition> must not have <state>s when the enclosing <process definition> contains a <service definition>. <procedure definition>s visible to more than one service must not contain a VIA construct.

The set of priorities associated to <continuous signal>s within the various <service definition>s of a <service decomposition> must not overlap.

Similar wellformedness rules apply for <signal route connect> as for <channel to route connection>.

If the enclosing <service decomposition> contains any <service signal route definition>s then for each <signal route identifier> in an <output> there must exist a service signal route originating from the enclosing service and connected to the signal route, and able to convey the signals denoted by the <signal identifier>s contained in the <output>.

If an <output> does not contain a VIA construct, then there must exist at least one communication path (either implicit to own service, or via (possibly implicit) service signal routes, and possibly signal routes and channels), originating from the service, that is able to convey the signals denoted by the <signal identifier>s contained in the <output>.

For each <priority output> there must exist at least one communication path (either implicit to own service, or via (possibly implicit service signal routes), originating from the service that is able to convey the signals denoted by the <priority signal identifier>s contained in the <priority output>.

<priority input> is only allowed in a <service body>. <priority output> is only allowed in a <service body> and in <procedure body>.

Concrete graphical grammar

```
<service diagram> ::=
        <frame symbol> contains
        {<service heading>
          {  {<service text area> }*
             {<graphical procedure reference>}*
             {<procedure diagram>}*
             {<macro diagram>}*
             <service graph area>  }set   }

<service heading> ::=
        SERVICE {<service name< | <service identifier>}

<service text area> ::=
        <text symbol> contains
         { <variable definition>
             | <data definition>
             | <timer definition>
             | <view definition>
             | <import definition>
             | <select definition>
             | <macro definition>  }*

<service graph area> ::=
```

<process graph area>

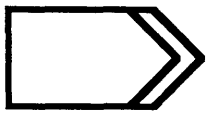
<priority input association area> ::=
<solid association symbol> **is connected to** <priority input area>

<priority input area> ::=
<priority input symbol> **contains** <priority input list>
is followed by <transition area>

<priority input symbol> ::=

<priority output area> ::=
<priority output symbol> **contains** <priority output body>

<priority output symbol> ::=



Semantics

The properties of a service are derived from the requirement that the <service decomposition> replacing a <process body> expresses the same behaviour as the <process body>.

Within a process instance there is a service instance for each <service definition> in the <process definition>. Service instances are components of the process instance, and cannot be manipulated (created, addressed or aborted) as separate objects. They share the input port and the expressions SELF, PARENT, OFFSPRING and SENDER of the process instance.

A service instance is a finite state machine, but it cannot run in parallel with other service instances of the process instance, i. e. within a process instance only one service instance can perform a transition at any one time.

In <priority output body> the construct TO SELF is implied. Priority signals are a special class of signals that have higher priority than ordinary signals. These signals can be sent only between service instances within the same process instance.

An input signal from the input port is given to the service instance that is able to receive that signal.

Model

a) *Transformation of definitions*

Local definitions within a <service definition> are transformed to the process level by replacing every occurrence of a name in the service by the same distinct new name. Every references to services in qualifiers disappear.
View definitions or import definitions containing the same view or import variable are merged into one view or import definition.

b) *Transformation of <service body>s*

The set of <service body>s is transformed into one <process body>. This may be done in several alternative ways. Here, a simple transformation is chosen, since the main objective is to define the service concept by strict concrete syntax. For practical reason a <service body> and a <process body> is regarded as a graph composed of states, transition strings between states, stop transition strings and one start transition string. A transition string is uniquely defined by a start state, an input and an end state.

1) *States*

A state in the resulting process graph is identified by a name-tuple. The dimension of the tuple is the number of service graphs. Each tuple component refers uniquely to one of the original services graphs, and the value of the tuple component is one of the state names of the referred service graph. The state names of the process graph will then be the set of tuples that is possible to construct using these rules. Example:

Given two service graphs and their states

f1: <a> <b>
f2: <A> <B> <C>

then the resulting process graph has the following states

<a.A> <a.B> <a.C> <b.A> <b.B> <b.C>

This state explosion can normally be reduced substantially, but this is not treated here.

2) *transition strings*

Each transition string in a service graph is copied into the process graph in one or more places. It is copied to connect each pair of state tuples that satisfies the following conditions:

- One component of the start state tuple refers to the start state of the transition string
- One component of the end state tuple refers to the end state of the transition string
- the other component values must be the same for both state tuples

Example:

In the previous example we have a transition string in f2 between <B> and <C>. In the resulting process graph, this transition string will connect <a.B> to <a.C> and <b.B> to <b.C>. This can be expressed more concisely (using the short hand notation of the concrete syntax):

<*.B> is transformed to <-.C>

3) *Start transition strings*

If one of the service graphs contains a start transition string, then this transition string is transformed into the start transition string of the process graph. The start transition string of the process graph leads to the state tuple having as components all the initial state names of the service graph.

4) *Stop transition strings*

Each transition leading to a <stop> is copied into the process graph and it is connected to each state tuple having one component that refers to the start state of the transition.

5) *Priority signals*

The priority signals are transformed as follows.

Each state of the resulting process graph is split into two states. Priority inputs to the original state are connected to the first state, all other inputs to the second state and are saved in the first state. The transition string leading to the original state is now leading to the first state. To this transition string is added the following action string:

- a unique token-value is generated and is assigned to the implicit variable SAME_TOKEN
- the implicit signal X_CONT is sent to SELF, carrying the token-value.

An input for the implicit signal X_CONT is added to the first state, followed by the following transition string:

A decision compares the received token-value with the value of SAME_TOKEN. If the values are equal, then a path leading to the second state is chosen, otherwise a part leading back to the first state.

Example

An example of a <process definition> containing a <service decomposition> is given below as well as the corresponding <service definition>s. This process has the same behaviour of the one given in Figure 2.9.9 in § 2.9.

```
PROCESS Game;
    FPAR Player pid;
    SIGNAL Proberers (integer);
    DCL A integer;

    SIGNALROUTE IR1 FROM Game_handler TO ENV WITH Score,Gameid;
    SIGNALROUTE IR2 FROM Game_handler TO ENV WITH Subscr,Endsubscr;
    SIGNALROUTE IR3 FROM ENV TO Game_handler WITH Result,Endgame;
    SIGNALROUTE IR4 FROM ENV TO Bump_handler WITH Probe;
    SIGNALROUTE IR5 FROM ENV TO Bump_handler WITH Bump;
    SIGNALROUTE IR6 FROM Bump_handler TO ENV WITH Lose,Win;
    SIGNALROUTE IR7 FROM Bump_handler TO Game_handler WITH Proberers;

    CONNECT R5 AND IR5;
    CONNECT R2 AND IR3,IR4;
    CONNECT R3 AND IR1,IR6;
    CONNECT R4 AND IR2;

    SERVICE Game_handler REFERENCED;
    SERVICE Bump_handler REFERENCED;

ENDPROCESS Game;
```

```
SERVICE Game_handler;

    /*The service handles a game with actions to start a game,
    to end a game, to keep track
    of the score and to communicate the score*/

    DCL Count integer;
    /*Counter to keep track of the score*/

    START;
        OUTPUT Subscr;
        OUTPUT Gameid TO Player;
        TASK Count:=0;
        NEXTSTATE STARTED;
    STATE STARTED;
        PRIORITY INPUT Proberers(A);
            TASK Count:=Count+A;
            NEXTSTATE _;
        INPUT Result;
            OUTPUT Score(Count) TO Player;
            NEXTSTATE _;
        INPUT Endgame;
            OUTPUT Endsubscr;
            STOP;
    ENDSTATE STARTED;
ENDSERVICE Game_handler;


SERVICE Bump_handler;

    /*The service has actions to register the bumps and
    to handle probes from the player.
    The probe result is sent to the player but also to the service Game_handler*/

    START;
        NEXTSTATE EVEN;
    STATE EVEN;
        INPUT Probe;
            OUTPUT Lose TO Player;
            PRIORITY OUTPUT Proberers(-1);
            NEXTSTATE _;
        INPUT Bump;
            NEXTSTATE ODD;
    ENDSTATE EVEN;
    STATE ODD;
        INPUT Bump;
            NEXTSTATE EVEN;
        INPUT Probe;
            OUTPUT Win TO Player;
            PRIORITY OUTPUT Proberers(+1);
            NEXTSTATE _;
    ENDSTATE ODD;
ENDSERVICE Bump_handler;
```

The same example in SDL/GR is shown in the following diagrams:

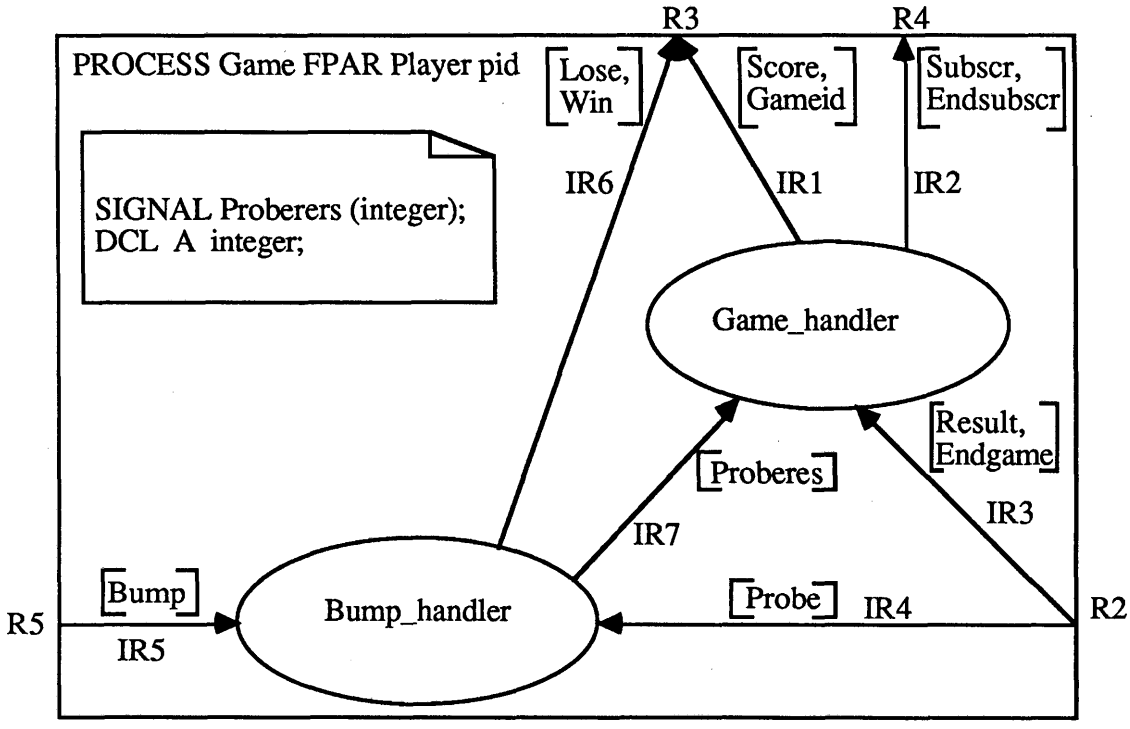


FIGURE 4.10.1

Example of a process diagram with service decomposition

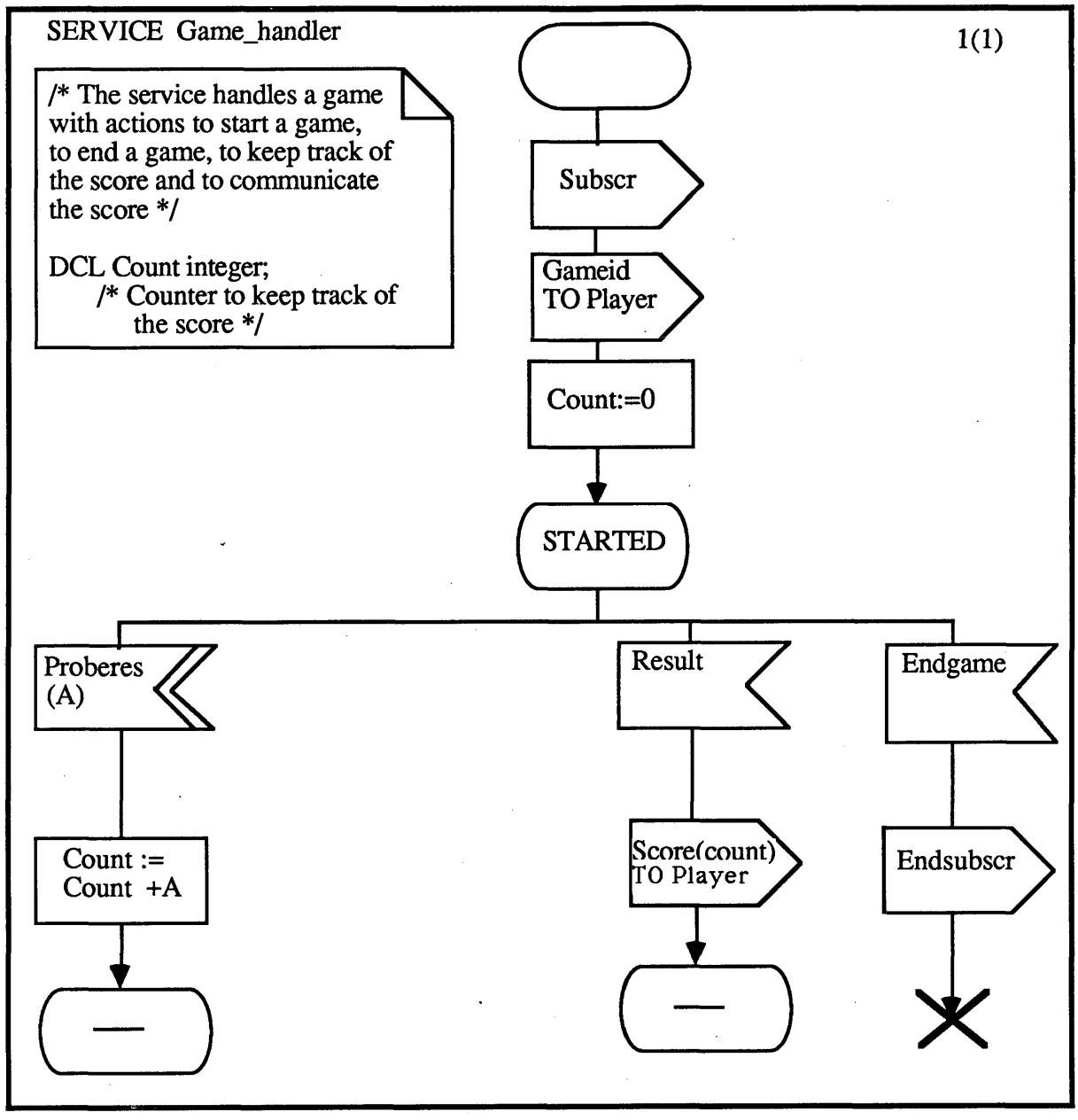


FIGURE 4.10.2

Example of a service diagram

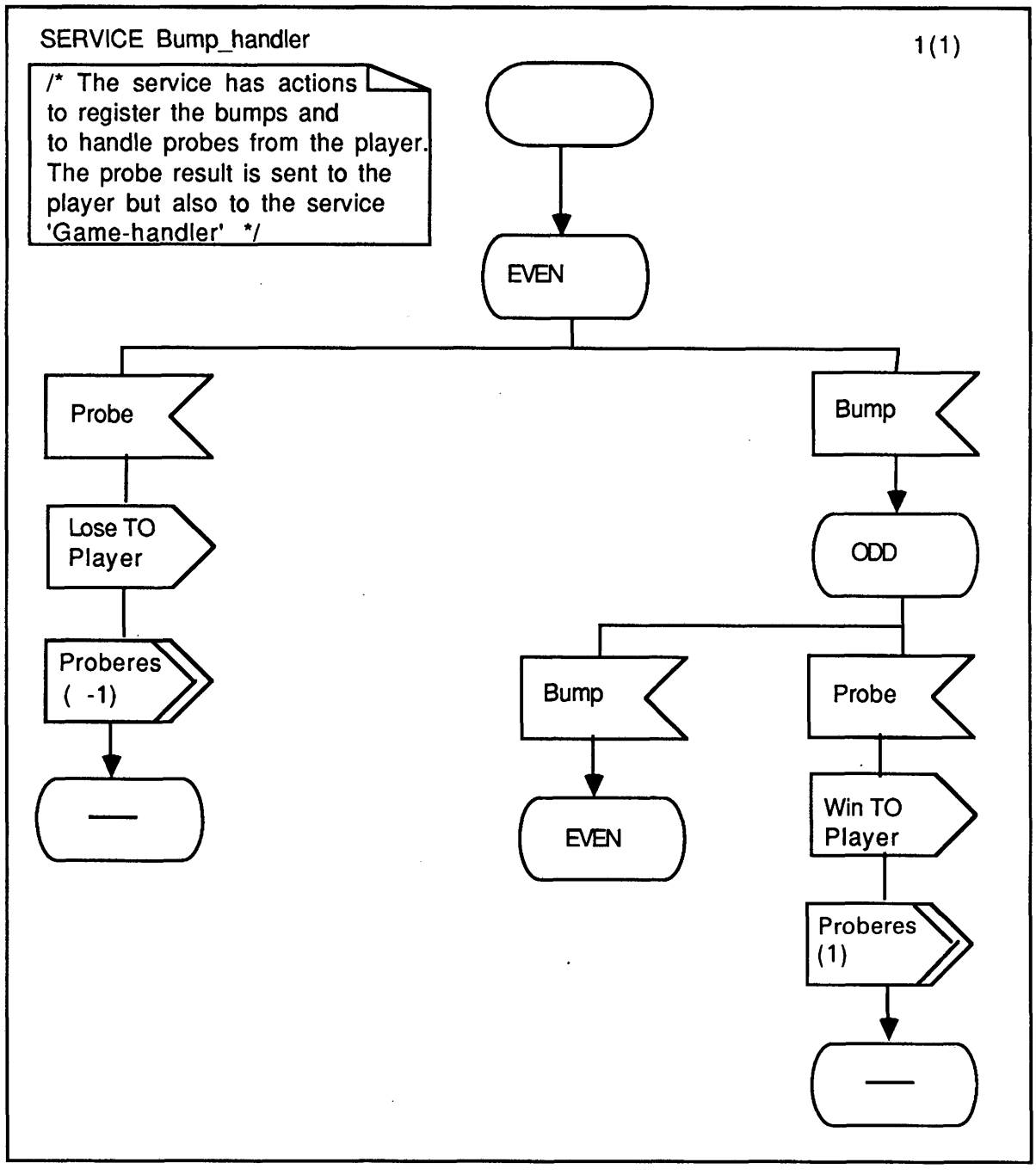


FIGURE 4.10.3

Example of a service diagram

Applying the rules from 1 to 4 of the transformation the process graph of Figure 4.10.4 is obtained; it still contains priority signals not yet transformed. Simplifying in an obvious way the transitions that contain priority signals and using the asterisk state concept, the same process of Figure 2.9.10 in § 2.9 can be obtained. (Note that the states EVEN and ODD correspond respectively to the states STARTED.EVEN and STARTED.ODD)

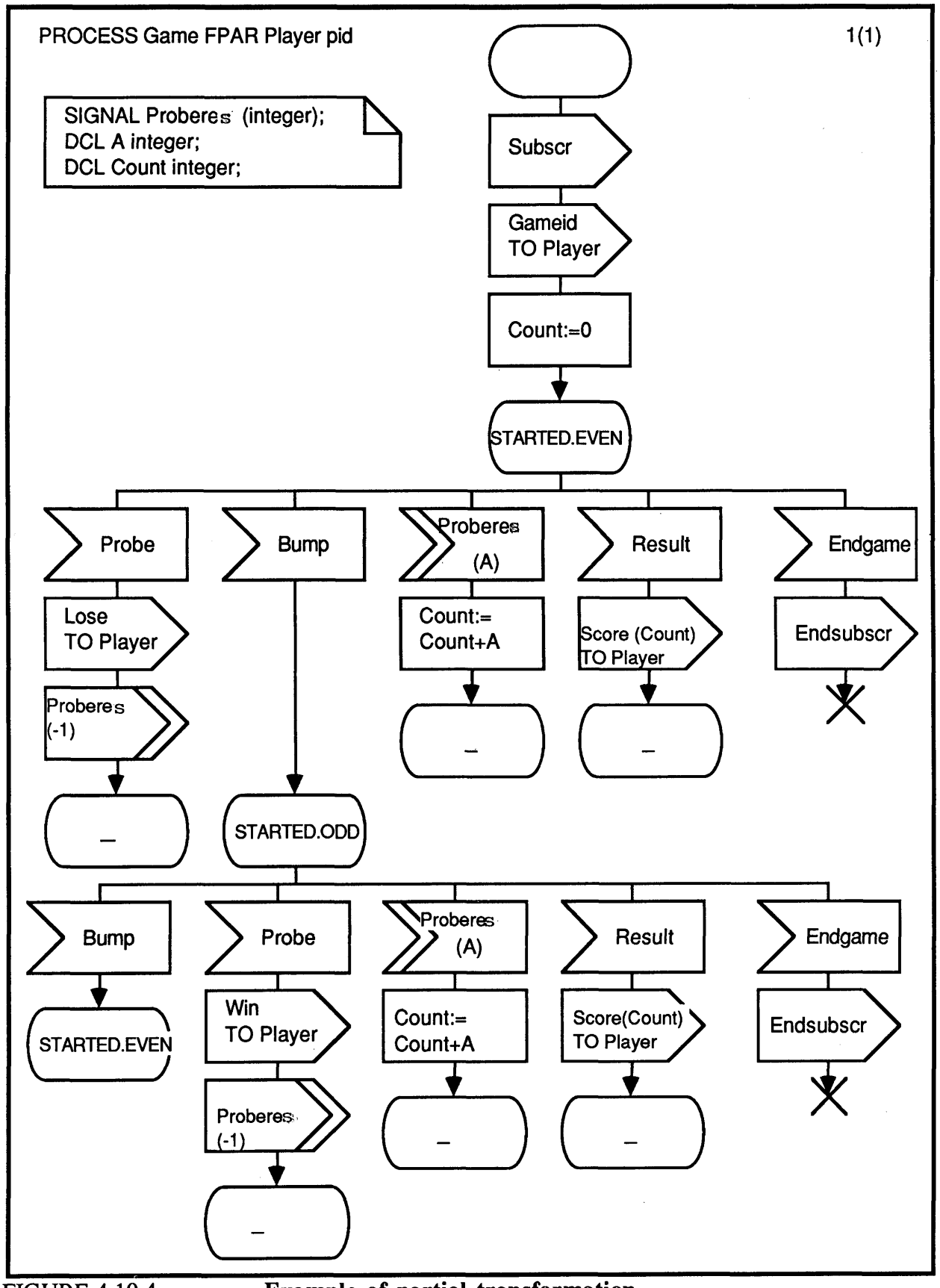


FIGURE 4.10.4 **Example of partial transformation**

4.11 *Continuous signal*

In describing systems with SDL, the situation may arise where a user would like to show that a transition is caused directly by a true value of a boolean expression. The model of achieving this is to evaluate the expression while in the state, and initiate the transition if the expression evaluates to true.A shorthand for this is called Continuous signal, which allows a transition to be initiated directly when a certain condition is fulfilled.

Concrete textual grammar

<continuous signal> ::=
PROVIDED <boolean expression> <end>
[PRIORITY <integer literal name> <end>] <transition>

The values of the <integer literal name>s in <continuous signal>s of a <state> must be distinct. The PRIORITY construct may be omitted only if the <state> contains exactly one <continuous signal>.

Concrete graphical grammar

<continuous signal association area> ::=
<solid association symbol> **is connected to** <continuous signal area>

<continuous signal area> ::=
<enabling condition symbol>
contains {<boolean expression> [[<end>] PRIORITY <integer literal name>]}
is followed by <transition area>

Semantics

The <boolean expression> in the <continuous signal> is evaluated upon entering the state to which it is associated, and while waiting in the state, any time no <stimulus> of an attached <input list> is found in the input port. If the value of the <boolean expression> is True, the transition is initiated. When the value of the <boolean expression> is True in more than one <continuous signal>s, then the transition to be initiated is determined by the <continuous signal> having the highest priority, that is the lowest value for <integer literal name>.

Model

The state with the name state_name containing <continuous signal>s is transformed to the following. This transformation requires two implicit variables n and newn. The variable n is initialised to 0. Furthermore an implicit signal emptyQ conveying an integer value is required.

1) All <nextstate>s which mention the state_name are replaced by JOIN 1;

2) The following transition is inserted:

```
1:  TASK  n:= n+1;
    OUTPUT  emptyQ (n) TO SELF;
    NEXTSTATE  state_name;
```

3) The following <input part> is added to the <state> state_name:

```
    INPUT  emptyQ (newn);
```

and a <decision> containing the <question>

```
        (newn=n)
```

4a) The false <answer part> contains

NEXSTATE state_name;

4b) The true <answer part> contains a sequence of <decision>s corresponding to the <continuous signal>s in priority order (higher priority is indicated by lower value of the <integer literal name>).
The False <answer part> contains the next <decision>, except for the last <decision> for which this <answer part> contains: JOIN 1;
Each true <answer part> of these <decision>s leads to the <transition> of the corresponding <continuous signal>.

Example

See § 4.12.

4.12 *Enabling condition*

In SDL the reception of a signal in a state immediately initiates a transition. The concept of Enabling condition makes it possible to impose an additonal condition for the initiation of a transition.

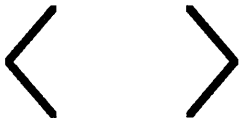
Concrete textual grammar

<enabling condition> ::=
PROVIDED <boolean expression> <end>

Concrete graphical grammar

<enabling condition area> ::=
<enbling condition symbol> **contains** <boolean expression>

<enabling condition symbol> ::=



Semantics

The <boolean expression> in the <enabling condition> is evaluated before entering the state in question, and any time the state is reentered through the arrival of a <stimulus>. In the case of multiple enabling conditions, these are evaluated sequentially in a non deterministic order before entering the state. The transformation model guarantees repeated reevaluation of the expression by sending additional <stimulus>s through the input port. A signal denoted in the <input list> which precedes the <enabling condition> can start a transition only if the value of the corresponding <boolean expression> is True. If this value is False, the signal is saved instead.

Model

The state with the name state_name containing <enabling condition>s is transformed to the following. This transformation requires two implicit variables n and newn. The variable n is initialised to 0. Furthermore an implicit signal emptyQ conveying an integer value is required.

1) All <nextstate>s which mention the state_name are replaced by JOIN 1;

2) The following transition is inserted:

   1: TASK n:= n+1;
   OUTPUT emptyQ (n) TO SELF;

   A number of decisions, each containing only one <boolean expression> corresponding to some <enabling condition> attached to the state, is added hierarchically in a non deterministic order such that all combination of truth values may be evaluated for all enabling conditions attached to the state.
   Each such combination leads to a new distinct state .

3) Each of these new states has a set of <input part>s consisting of a copy of these <input part>s of the state without enabling conditions plus the <input part>s for which the <enabling condition>'s <boolean expression>s evaluated to true for this state.
   The <stimulus>s for the remaining <input part>s constitute the <save list> for a new <save part> attached to this state. The <save part>s of the original state are also copied to this new state.

4) Add to each of the new states:

   INPUT emptyQ (newn);

   A <decision> containing the <question> (newn=n);
   The false <answer part> contains a <nextstate> back to this same new state.

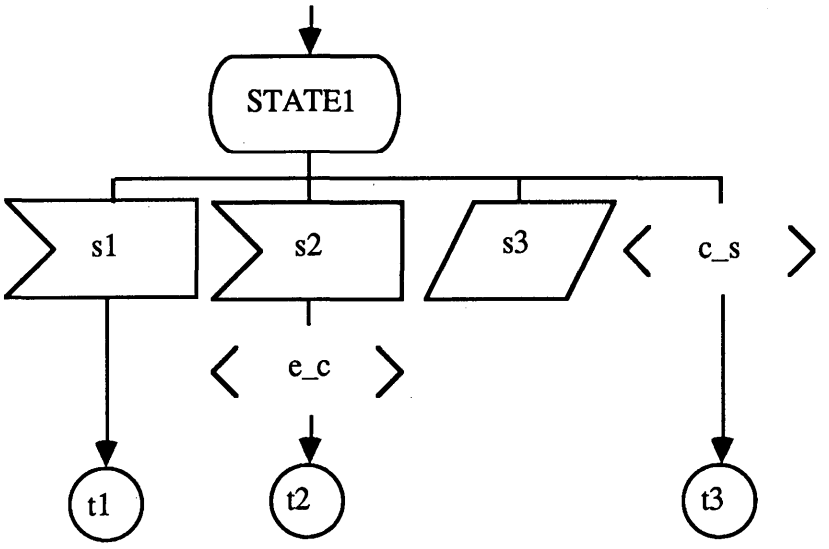
5) The true <answer part> contains a JOIN 1;

6) If <continuous signal>s and <enabling condition>s are used in the same <state>, evaluations of the <boolean expression>s from <continuous signal>s are done by replacing step 5 of the model for <enabling condition> with step 4b of the model for <continuous signal>.

Example

An example illustrating the transformation of continuous signal and enabling condition appearing in a state is given below.

Note in the example that the connector ec has been introduced for convenience. It is not part of the transformation model.



is transformed into

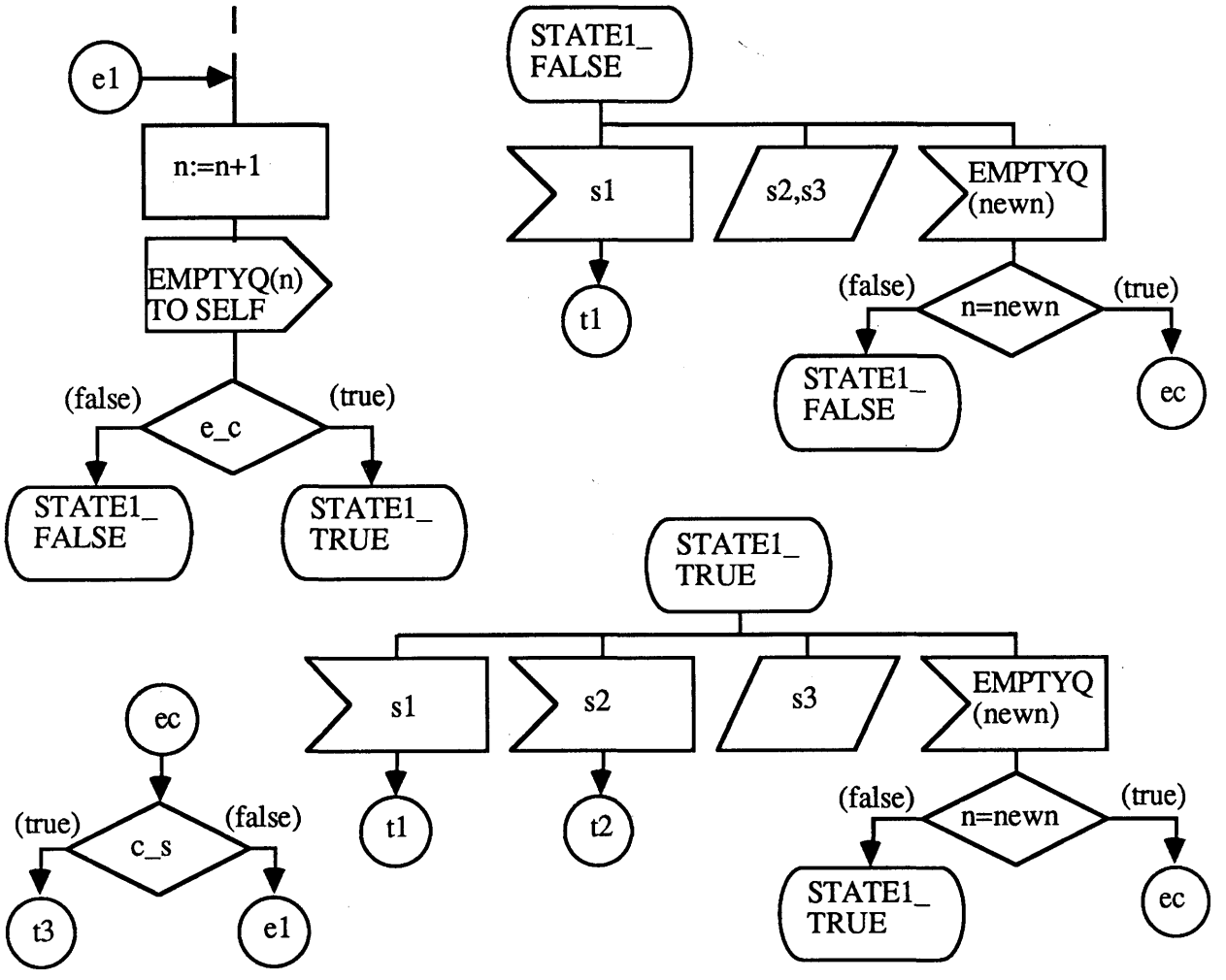


FIGURE 4.12.1
Transformation of continuos signal and enabling condition in the same state

4.13 *Imported and Exported value*

In SDL a variable is always owned by, and local to, a process instance. Normally the variable is visible only to the process instance which owns it, though it may be declared as a shared value (see §2) which allows other process instances in the same block to have access to the value of the variable. If a process instance in another block needs to access the value of a variable, a signal interchange with the process instance owning the variable is needed.

This can be achieved by the following shorthand notation, called imported and exported value. The shorthand notation may also be used to export values to other process instances within the same block, in which case it provides an alternative to the use of shared values.

Concrete textual grammar

<import definition> ::=
IMPORTED <import name> {, <import name> }* <sort>
{, <import name> {, <import name> }* <sort>}* <end>

<import expression> ::=
IMPORT (<import identifier> [, <pid expression>])

<export> ::=
EXPORT (<variable identifier> {, <variable identifier> }*)

Concrete graphical grammar

<export area>::=
<task symbol> **contains** <export>

Semantics

The process instance which owns a variable whose values are exported to other process instances is called the exporter of the variable. Other process instances which use these values are known as importers of the variable. The variable is called exported variable.

A process instance may be both importer and exporter, but it cannot import from or export to the environment.

a) *Export operation*

Exported variables have the keyword EXPORTED in their <variable definition>s, and have an implicit copy to be used in import operations.

An export operation is the execution of an <export> by which an exporter discloses the current value of an exported variable . An export operation causes the storing of the current value of the exported variable into its implicit copy.

b) *Import operation*

For each <import definition> in an importer there is a set of implicit variables, all having the name and sort given in the <import definition>. These implicit variables are used for the storage of imported values.

An import operation is the execution of an <import expression> by which an importer accesses the value of an exported variable. The value is stored in an implicit variable denoted by the <import identifier> in the <import expression>. The exporter containing the exported variable is specified by the <pid expression> in the <import expression>. If no <PId expression> is specified then there should be only one instance exporting that variable.The association between the exported variable in the exporter and the implicit variable in the importer is specified by having the same <identifier> in the <export> and in the <import expression>. In addition, the exported variable and the implicit variable must have the same sort.

Model

An import operation is modeled by exchange of signals. These signals are implicit and are conveyed on implicit channels and signal routes. The importer sends a signal to the exporter, and waits for the reply. In response to this signal the exporter sends a signal back to the importer with the value contained in the implicit copy of the exported variable.

If a default assignement is attached to the export variable or if the export variable is initiated when it is defined, then also the implicit copy is initiated and with the same value as the export variable.

There are two implicit <signal definition>s for each combination of <import name> and <sort> contained in all <import definition>s in a system definition. The <signal name>s in these <signal definition>s is denoted by *xtQUERY* respectively *xtREPLY*, where *x* denotes an <import name> and *t* denotes a <sort>. The implicit copy of the exported variable is denoted by *imcx*.

a) *Importer*

The <import expression> 'IMPORT (x, pidexp)' is transformed to the following:

```
OUTPUT xtQUERY TO pidexp;
Wait in state xtWAIT, saving all other signals;
INPUT xtREPLY (x);
Replace the <import expression> by x, (the <name> of the implicit variable);
```

If an <import expression> occurs more than once in an <expression>, then a separate implicit variable with the same <name> is used for each occurence.

b) *Exporter*

To all <state>s, including implicit states, of the exporter the following <input part> is added:

```
INPUT xtQUERY;
OUTPUT xtREPLY (imcx) TO SENDER;
/* next state the same */
```

The <export> 'EXPORT (x)' is transformed to the following:

```
TASK imcx := x;
```

5 Data in SDL

5.1 *Introduction*

This introduction gives an outline of the formal model used to define data types and information on how the rest of § 5 is structured.

In a specification language, it is essential to allow data types to be formally described in terms of their behaviour, rather than by composing them from provided primitives, as in some programming languages. The latter approach invariably involves a particular implementation of the data type, and hence restricts the freedom available to the implementer to choose appropriate representations of the data type. The abstract data type approach allows any implementation providing that it is feasible and correct with respect to the specification.

5.1.1 *Abstraction in data types*

All data used in SDL is based on abstract data types which are defined in terms of their abstract properties rather than in terms of some concrete implementation. Examples of defining abstract data types are given in § 5.6 which defines the predefined data facilities of the language.

Although all data types are abstract, and the predefined data facilities may even be overridden by the user, SDL attempts to provide a set of predefined data facilities which are familiar in both their behaviour and syntax. The following are predefined:

a) Boolean

b) Character

c) String

d) Charstring

e) Integer

f) Natural

g) Real

h) Array

i) Powerset

j) PId

k) Duration

l) Time.

The structured sort concept (STRUCT) can be used to form composite objects.

5.1.2 *Outline of formalisms used to model data*

Data is modelled by an initial algebra. The algebra has designated sorts, and a set of operators mapping between the sorts. Each sort is the collection of all the possible values which can be generated by the related set of operators. Each value can be denoted by at least one expression in

the language containing only literals and operators (except in the special case of PId values). Literals are a special case of operators without arguments.

The sorts and operators, together with the behaviour (specified by algebraic rules) of the data type, form the properties of the data type. A data type is introduced in a number of partial type definitions, each of which defines a sort and operators and algebraic rules associated with that sort.

The keyword NEWTYPE introduces a partial type definition which defines a distinct new sort. A sort can be created with properties inherited from another sort, but with different identifiers for the sort and operators.

Introduction of a syntype nominates a subset of the values of an already existing sort.

A generator is an incomplete NEWTYPE description: before it assumes the status of a sort, it must be instantiated by providing the missing information.

Some operators map onto the sort, and so produce (possibly new) values of the sort. Other operators give meaning to the sort by mapping onto other defined sorts. Many operators map onto the Boolean sort from other sorts, but it is strictly prohibited for these operators to extend the Boolean sort.

In SDL a function is known as a passive operator and can have no effect on the values associated with variables given as parameters. SDL also defines assignment which can change the values associated with variables.

5.1.3 *Terminology*

The terminology used in § 5 or the data model is chosen to be in harmony with published work on initial algebras. In particular "data type" is used to refer to a collection of sorts plus a collection of operators associated with those sorts and the definition of properties of these sorts and operators by algebraic equations. A "sort" is a set of values with common characteristics. An "operator" is a relation between sorts. An "equation" is a definition of equivalence between terms of a sort. A value is a set of equivalent terms. An "axiom" is an equation which defines a Boolean value to be equivalent to True. However, "axioms" is used as a term for "axiom"s or "equation"s, and an "equation" can be an "axiom".

5.1.4 *Division of text on data*

The initial algebra model used for data in SDL is described in a way which allows most of the data concepts to be defined in terms of a data kernel of the SDL abstract data language.

The text of § 5 is divided into this introduction (§ 5.1), the data kernel language (§ 5.2), the initial algebra model (§ 5.3), passive use of data (§ 5.4), active use of data (§ 5.5) and predefined data (§ 5.6).

The data kernel language defines the part of data in SDL which corresponds directly with the underlying initial algebra approach.

The text on initial algebra gives a more detailed introduction to the mathematical basis of this approach. This is formulated in a more precise mathematical way in appendix I.

The passive use of SDL includes the implicit and shorthand features of SDL data which allow its use for the definition of abstract data types. It also includes the interpretation of expressions which do not involve values assigned to variables. These "passive" expressions correspond to functional use of the language.

The active use of data extends the language to include assignment. This includes assignment to use of and initialisation of variables. When SDL is used to assign to variables or to access the values in variables, it is said to be used actively. The difference between active and passive expressions is that the value of a passive expression is independent of when it is interpreted, whereas an active expression may be interpreted as different values depending on the current values associated with variables or the current system state.

The final topic is predefined data.

5.2 *The data kernel language*

The data kernel can be used to define abstract data types.

More convenient constructs for defining data types can be defined in terms of the constructs defined for the data kernel, except where the concepts of assignment to a variable are needed. (The concepts of errors and syntypes could be defined in terms of the kernel but in § 5.4.1.7 and § 5.4.1.9 alternative, more concise, definitions are used).

5.2.1 *Data type definitions*

At any point in an SDL specification there is an applicable data type definition. The data type definition defines the validity of expressions and the relationship between expressions. The definition introduces operators and sets of values (sorts).

There is not a simple correspondence between the concrete and abstract syntax for data type definitions since the concrete syntax introduces the data type definition incrementally with emphasis on the sorts (see also § 5.3).

The definitions in the concrete syntax are often interdependent and cannot be separated into different scope units. For example

```
NEWTYPE even LITERALS 0;
    OPERATORS  plusee     : even, even  -> even;
               plusoo     : odd, odd    -> even;
    AXIOMS     plusee(a,0)     == a;
               plusee(a,b)     == plusee(b,a);
               plusoo(a,b)     == plusoo(b,a);
    ENDNEWTYPE even  COMMENT 'even "numbers" with plus–depends on odd';
NEWTYPE odd LITERALS 1;
    OPERATORS  plusoe     : odd, even   -> odd;
               pluseo     : even, odd   -> odd;
    AXIOMS     plusoe(a,0)     == a;
               pluseo(a,b)     == plusoe(b,a);
    ENDNEWTYPE odd; /*odd "numbers" with plus - depends on even*/
```

Each data type definition is complete; there are no references to sorts or operators which are not included in the data type definition which applies at a given point. Also a data type definition must not invalidate the semantics of a data type definition in the immediately surrounding scope unit. A data type in an enclosed scope unit only enriches operators of sorts defined in the outer scope unit. A value of a sort defined in a scope unit may be freely used and passed between or from hierarchically lower scope units. Since predefined data is defined at system level the predefined sorts (for example Boolean and Integer) may be freely used throughout the system.

Abstract grammar

Data-type-definition	::	*Type-name* *Type-union* *Sorts* *Signature*–**set** *Equations*
Type-union	=	*Type-identifier*–**set**
Type-identifier	=	*Identifier*
Sorts	=	*Sort-name*-**set**
Type-name	=	*Name*
Sort-name	=	*Name*
Equations	=	*Equation*–**set**

Within a *data type definition* for each *Sort* there must be at least one *Signature* with a *Result* (see § 5.2.2) which is the same as the *Sort*.
A *data type definition* must not add new values to any *sort* of the data type identified by the *type union*.

If one *term* (see § 5.2.3) is non-equivalent to another *term* according to the data type identified by the *type union* of a *data type definition*, then these *terms* must not be defined to be equivalent by the *data type definition*.

In addition the two Boolean *terms* True and False must not be (directly or indirectly) defined to be equivalent (see § 5.4.3.1). Also it is not allowed to reduce the number of values for the predefined sort PId.

Note — The abstract syntax allows more than one type identity for a *type union* to harmonise with the more general class of algebras used for the underlying model - in SDL only one type is referenced because in the concrete syntax the visible data type is implicitly defined by the surrounding <scope unit class>; therefore <type union> is only referenced in the abstract syntax and is either the *type identifier* of the surrounding scope unit or in the case of a <system definition> an empty set.

Concrete textual grammar

<partial type definition> ::=
NEWTYPE <sort name> [<extended properties>] <properties expression>
ENDNEWTYPE [<sort name>]

<properties expression> ::=
<operators> [AXIOMS <axioms>] [<literal mapping>] [<default assignment>]

The optional <extended properties>, <literal mapping> and <default assignment> are not part of the data kernel and are defined in sections § 5.4.1, § 5.4.1.15 and § 5.5.3.3 respectively.

The *data type definition* is represented by the collection of all the <partial type definition>s in the current <scope unit class> combined with the *data type definition* identified by the *type union* of the surrounding <scope unit class>. The type name of a <data type definition> is implied and does

not have a concrete syntax representation. The *type identifier* of a *type union* is implied to be the identity of the *data type definition* of the surrounding scope unit.

The following <scope unit class>s (see § 2.2.2) each represent an item in the abstract syntax which contains a *data type definition* : <system definition>, <block definition>, <process definition>, <procedure definition>, <channel substructure definition> or <block substructure definition> or the corresponding diagrams in graphical syntax. The <partial type definition> in a <service definition> represents part of the *data type definition* in the enclosing <process definition> of the <service definition> (see § 4.10).

The *sorts* for a <scope unit class> are represented by the set of <sort name>s introduced by the set of <partial type definition>s of the <scope unit class>.

The *signature* set and *equations* for a <scope unit class> are represented by the <properties expression>s of the <partial type definition>s of the <scope unit class>.

The <operators> of a <properties expression> represents part of the *signature* set in the abstract syntax. The complete *signature* set is the union of the *signature* sets defined by the <partial type definition>s in the <scope unit class>.

The <axioms> of a <properties expression> represents part of the *equation* set in the abstract syntax. The *equations* is the union of the *equation* sets defined by the <partial type definition>s in the <scope unit class>.

The predefined data sorts have their implicit <partial type definition>s at the system level.

If a <sort name> is given after the keyword ENDNEWTYPE then it must be the same as the <sort name> given after the keyword NEWTYPE.

Semantics

The data type definition defines a data type. A data type has a set of type properties, that is: a set of sorts, a set of operators and a set of equations.

The properties of data types are defined in the concrete syntax by partial type definitions. A partial type definition does not introduce all the properties of a data type but only partially defines some of the properties related to the sort introduced in the partial type definition. The complete properties of a data type are found by considering the combination of all partial type definitions which apply within the scope unit containing the data type definition.

A sort is a set of data values. Two different sorts have no values in common.

The data type definition is formed from the data type definition of the scope unit defining the current scope unit taken in conjunction with the sorts, operators and equations defined in the current scope unit. The system definition contains the definition of the predefined data sorts.

Except within a <partial type definition>, a <signal refinement> or a <service definition>, the data type definition which applies at any point is the data type defined for the scope unit immediately enclosing that point. Within a <partial type definition> or a <signal refinement> the data type definition which applies is the data type definition of the scope unit enclosing the <partial type definition> or <signal refinement> respectively. Within a <service definition> it is the *data type definition* of the enclosing <process definition> of the <service definition> which applies (see §4.10).

The set of sorts of a data type is the set of sorts introduced in the current scope unit plus the set of

sorts of the data type identified by the type union. The set of operators of a data type is the set of operators introduced in the current scope unit plus the set of operators of the data type identified by the type union. The set of equations of a data type is the set of equations introduced in the current scope unit plus the set of equations of the data type identified by the type union.

Each sort introduced in a data type definition has an identifier which is the name introduced by a partial type definition in the scope unit qualified by the identifier of the scope unit.

A data type has an identifier which is the unique type name in the abstract syntax qualified by the identity of the scope unit. There is no name for a data type in the concrete syntax.

Example

```
NEWTYPE telephone
        /* operators and construction of values defined elsewhere*/
ENDNEWTYPE telephone;
```

5.2.2 *Literals and parameterised operators*

Abstract grammar

Signature	=	*Literal-signature* | *Operator-signature*
Literal-signature	::	*Literal-operator-name* *Result*
Operator-signature	::	*Operator-name* *Argument-list* *Result*
Argument-list	=	*Sort-reference-identifier* $^{+}$
Result	=	*Sort-reference-identifier*
Sort-reference-identifier	=	*Sort-identifier* | *Syntype-identifier*
Literal-operator-name	=	*Name*
Operator-name	=	*Name*
Sort-identifier	=	*Identifier*

Syntypes and *syntype identifiers* are not part of the kernel (see § 5.4.1.9).

Concrete textual grammar

```
<operators> ::=
        [ <literal list> ] [ <operator list> ]

<literal list> ::=
        LITERALS <literal signature> {, <literal signature> }* [ <end> ]
```

<literal signature> ::=
        <literal operator name>
    | <extended literal name>

<operator list> ::=
        OPERATORS
            <operator signature> { <end> <operator signature> }* [<end>]

<operator signature> ::=
        <operator name> : <argument list> -> <result>
    | <ordering>

<operator name> ::=
        <operator name>
    | <extended operator name>

<argument list> ::=
        <argument sort> { , <argument sort> }*

<argument sort> ::=
        <extended sort>

<result> ::=
        <extended sort>

<extended sort> ::=
        <sort>
    | <generator sort>

<sort> ::=
        <sort identifier>
    | <syntype>

The alternatives <extended operator name>, <extended literal name>, <ordering>, <generator sort>, <generator sort> and <syntype> are not part of the data kernel and are defined in § 5.4.1, § 5.4.1, § 5.4.1.8, § 5.4.1.12.1, § 5.4.1.12.1 and § 5.4.1.9 respectively.

Literals are introduced by <literal signatures>s listed after the keyword LITERALS. The *result* of a *literal signature* is the sort introduced by the <partial type definition> defining the literal.

Each <operator signature> in the list of <operator signature>s after the keyword OPERATORS represents an *operator signature* with an *operator name*, an *argument list* and a *result*.

The <operator name> corresponds to an *operator name* in the abstract syntax which is unique within the defining scope unit even though the name may not be unique in the concrete syntax.

The unique *Operator-name* or *Literal-operator-name* in the abstract syntax is derived from

a) the <operator name> (or <literal operator name>), plus

b) the list of argument sort identifiers, plus

c) the result sort identifier, plus

d) the sort identifier of the partial type definition in which the <operator name> (or <literal operator name>) is defined.

Whenever an <operator identifier> is specified then the unique *operator name* in *operator identifier* is derived in the same way with the list of argument sorts and the result sort derived from context. Two operators with the same <name> which differ by one or more of the argument or result sorts have different *names*.
Each <argument sort> in an <argument list> represents a *sort reference identifier* in an *argument list*. A <result> represents the *sort reference identifier* of a *result*.

Wherever a <qualifier> of an <operator identifier> (or <literal operator identifier>) contains a <path item> with the keyword TYPE, then the <sort name> after this keyword does not form part of the *Qualifier* of the *Operator-identifier* (or *Literal-operator-identifier*) but is used to derive the unique *Name* of the *Identifier*. In this case the *Qualifier* is formed from the list of <path item>s preceding the keyword TYPE.

Semantics

An operator is "total" which means that application of the operator to any list of values of the argument sorts denotes a value of the result sort.

An operator signature defines how the operator may be used in expressions. The operator signature is the operator identity plus the list of sorts of the arguments and the sort of the result. It is the operator signature which determines whether an expression is a valid expression in the language according to the rules required for matching the sorts of argument expressions.

An operator with no argument is called a literal.

A literal represents a fixed value belonging to the result sort of the operator.

An operator has a result sort which is the sort identified by the result.

Note — As guidelines: an <operator signature> should mention the sort introduced by the enclosing <partial type definition> as either an <argument> or a <result>.

Example 1

```
LITERALS      free, busy ;
```

Example 2

```
OPERATORS
      findstate :     Telephone -> Availability;
```

Example 3

```
LITERALS      empty_list
OPERATORS     add_to_list     : list_of_telephones, telephone -> list_of_telephones;
              sub_list        : list_of_telephones, telephone -> list_of_telephones
```

5.2.3 *Axioms*

The axioms determine which terms represent the same value. From the axioms in a data type definition the relationship between argument values and result values of operators is determined and hence meaning is given to the operators. Axioms are either given as Boolean axioms or in the form

of algebraic equivalence equations.

Abstract grammar

Equation	=	*Unquantified-equation* | *Quantified-equations* | *Conditional-equation* | *Informal-text*
Unquantified-equation	::	*Term* *Term*
Quantified-equations	::	*Value-name*–**set** *Sort-identifier* *Equations*
Value-name	=	*Name*
Term	=	*Ground-term* | *Composite-term* | *Error-term*
Composite-term	::	*Value-identifier* | *Operator-identifier Term*$^{+}$ | *Conditional-composite-term*
Value-identifier	=	*Identifier*
Operator-identifier	=	*Identifier*
Ground-term	::	*Literal-operator-identifier* | *Operator-identifier Ground-term*$^{+}$ | *Conditional-ground-term*
Literal-operator-identifier	=	*Identifier*

The alternatives *Conditional-composite-term* and *Conditional-ground-term* in the rules *Composite-term* and *Ground-term* respectively are not part of the data kernel, although the equations containing these terms may be replaced by semantically equivalent equations written in the kernel language (see § 5.4.1.6). The alternative *error term* in the rule *term* is not part of the data kernel and is defined in § 5.4.1.7.

The definitions of *informal text* and *conditional equations* are given in § 2.2.3 and § 5.2.4 respectively.

Each *term* (or *ground term*) in the list of terms after an *operator identifier* must have the same sort as the corresponding (by position) sort in the *argument list* of the *operator signature.*

The two *terms* in an *unquantified equation* must be of the same sort.

Concrete textual grammar

<axioms> ::=
<equation> { <end> <equation>}* [<end>]

<equation> ::=
<unquantified equation>
| <quantified equations>
| <conditional equation>
| <informal text>

<quantified equations> ::=
<quantification> (<axioms>)

<quantification> ::=
FOR ALL <value name> { , <value name> }* IN <extended sort>

<unquantified equation> ::=
<term> == <term>
| <Boolean axiom>

<term> ::=
<ground term>
| <composite term>
| <error term>
| <spelling term>

<composite term> ::=
<value identifier>
| <operator identifier> (<composite term list>)
| (<composite term>)
| <extended composite term>

<composite term list> ::=
<composite term> { , <term> }*
| <term> , <composite term list>

<ground term> ::=
<literal identifier>
| <operator identifier> (<ground term> { , <ground term> }*)
| (<ground term>)
| <extended ground term>

<literal identifier> ::=
<literal operator identifier>
| <extended literal identifier>

The alternatives <Boolean axiom> of rule <unquantified equation>, <error term> and <spelling term> of rule <term>, <extended composite term> of rule <composite term>, <extended ground term> of rule <ground term>, and <extended literal identifier> of rule <literal identifier> are not part of the data kernel and are defined in § 5.4.1.5, § 5.4.1.7, § 5.4.1.15, § 5.4.1, § 5.4.1, and § 5.4.1 respectively.

The <sort> in a <quantification> represents the *sort identifier* in *quantified equations* . The <value name>s in a <quantification> represents the *value name* set in *quantified equations* .

A <composite term list> represents a *term* list. An *operator identifier* followed by a *term* list is only a *composite term* if the *term* list contains at least one *value identifier*.

An <identifier> which is an unqualified name appearing in a <term> represents

a) an *operator identifier* if it precedes an open round bracket (or it is an <operator name> which is an <extended operator name> – see § 5.4.1), otherwise

b) a *value identifier* if there is a definition of that name in a <quantification> of <quantified equations> enclosing the <term> of a suitable sort for the context, otherwise

c) a *literal operator identifier* if there is a visible literal with that name of a suitable sort for the context, otherwise

d) a *value identifier* which has an implied *quantified equation* in the abstract syntax for the <unquantified equation>.

Two or more occurrences of the same unbound <value identifier> in an <equation> imply only one *quantification.*

An *operator identifier* is derived from the context so that if the <operator name> is overloaded (that is the same <name> is used for more than one operator) then it will be the *operator name* which identifies a visible operator with the same name and the argument sorts and result sort consistent with the operator application. If the <operator name> is overloaded then it may be necessary to derive the argument sorts from the arguments and the result sort from context in order to determine the *operator name.*

Within one <unquantified equation> there must be exactly one sort for each implicitly quantified value identifier which is consistent with all its uses.

It must be possible to bind each unqualified <operator identifier> or <literal operator identifier> to exactly one defined *operator identifier* or *literal operator identifier* which satisfies the conditions in the construct in which the <identifier> is used. That is the binding shall be unique.

Note — As guidelines: an axiom should be relevant to the sort of the enclosing partial type definition by mentioning an operator or literal with a result of this sort or an operator which has an argument of this sort; an axiom should be defined only once.

Semantics

Each equation is a statement about the algebraic equivalence of terms. The left hand side term and right hand side term are stated to be equivalent so that where one term appears, the other term may be substituted. When a value identifier appears in an equation then it may be simultaneously substituted in that equation by the same term for every occurrence of the value identifier. For this substitution the term may be any ground term of the same sort as the value identifier.

Value identifiers are introduced by the value names in quantified equations. A value identifier is used to represent any data values belonging to the sort of the quantification. An equation will hold if the same value is simultaneously substituted for every occurrence of the value identifier in the equation regardless of the value chosen for the substitution.

A ground term is a term which does not contain any value identifiers. A ground term represents a particular, known value. For each value in a sort there exists at least one ground term which represents that value.

If any axioms contain informal text then the interpretation of expressions is not formally defined by SDL but may be determined from the informal text by the interpreter. It is assumed that if informal text is specified the equation set is known to be incomplete, therefore complete formal specification has not been given in SDL.

A value name is always introduced by quantified equations in the abstract syntax, and the corresponding value has a value identifier which is the value name qualified by the sort identifier of the enclosing quantified equations. For example

FOR ALL z,z IN X (FOR ALL z IN X ...))

introduces only one value identifier named z of sort X.

In the concrete syntax it is not allowed to specify a qualifier for value identifiers.

Each value identifier introduced by quantified equations has a sort which is the sort identified in the quantified equations by the *sort reference identifier*. The sort of the implied quantifications is the sort required by the context(s) of the occurrence of the unbound identifier. If the contexts of a value identifier which has implied quantification allow different sorts then the identifier is bound to a sort which is consistent with all its uses in the equation.

A term has a sort which is the sort of the value identifier or the result sort of the (literal) operator.

Unless it can be deduced from the equations that two literals denote the same value then each literal denotes a different value.

Example 1

FOR ALL b IN logical (eq(b,b)==T)

Example 2

neq(T,F)==T; neq(T,T) == F;
neq(F,T)==T; neq(F,F) == F;

Example 3

eq(b, b) == T;
eq(F, eq(T,F)) == T;
eq(eq(b,a),eq(a,b)) == T;

5.2.4 *Conditional equations*

A conditional equation allows the specification of equations which only hold when certain restrictions hold. The restrictions are written in the form of simple equations.

Abstract grammar

Conditional-equation	::	*Restriction*–**set** *Restricted-equation*
Restriction	=	*Unquantified-equation*
Restricted-equation	=	*Unquantified-equation*

Concrete textual grammar

<conditional equation> ::=
<restriction> { , <restriction> }* ==> <restricted equation>

<restricted equation> ::=
<unquantified equation>

<restriction> ::=
<unquantified equation>

Semantics

A restricted equation defines that terms denote the same value only when any value identifier in the restricted equations denotes a value which can be shown from other equations to satisfy the restriction. A value will satisfy a restriction only if the restriction can be deduced from other equations for this value.

The semantics of a set of equations for a data type which includes conditional equations are derived as follows:-

a) Quantification is removed by generating every possible ground term equation which can be derived from the quantified equations. As this is applied to both explicit and implicit quantification a set of unquantified equations in ground terms only is generated.

b) Let a conditional equation for which all the restrictions (in ground terms only) can be proved to hold from unquantified equations which are not restricted equations be called a provable conditional equation. If there exists a provable conditional equation, then it is replaced by the restricted equation of the provable conditional equation.

c) If there are conditional equations remaining in the set of equations and none of these conditional equations are a provable conditional equation, then these conditional equations are deleted, otherwise return to step (b).

d) The remaining set of unquantified equations defines the semantics of the data type.

Example

z /= 0 == True ==> (x/z)*z==x

5.3 *Initial algebra model (informal description)*

The definition of data in SDL is based on the data kernel defined in §5.2. Operators and values need to be given some further meaning in addition to the former definition so interpretation can be given to expressions. For example expressions used in continuous signals, enabling conditions, procedure calls, output actions, create requests, assignment statements, set and reset statements, export statements, import statements, decisions, and viewing.

The necessary additional meaning is given to expressions by using the initial algebra formalism which is explained in § 5.3.1 to § 5.3.6 below[1)].

At any point in an SDL specification the last data type hierarchically defined will apply, but there will be a set of sorts visible. The set of sorts will be the union of all sorts at levels hierarchically above the place in question as explained in §5.2.

[1)]The text of § 5.3.1 to § 5.3.6 has been agreed between ISO and CCITT as a common informal description of the initial algebra model for abstract data types. As well as appearing in this recommendation this text (with appropriate typographical and numbering changes) is also an annex to ISO IS8807.

(In this section the symbol = is used as an equation equivalence symbol whereas in SDL symbol == is used for equation equivalence so that the symbol = can be used for the equality operator. The symbol = is used in this section as it is the conventional symbol used in published work on initial algebras.)

5.3.1 *Introduction*

The meaning and interpretation of data based on initial algebra is explained in three stages:

a) Signatures

b) Terms

c) Values

5.3.1.1 *Representations*

The idea that different notations can represent the same concept is commonplace. For instance it is generally accepted that positive Arabic numbers (1,2,3,4,...) and Roman numerals (I,II,III,IV,...) represent the same set of numbers with the same properties. As another example it is quite usual to accept that prefix functional notational (plus(1,1)), infix notation (1+1) and reverse polish notation (1 1 +) can all represent the same operator. Furthermore different users may use different names (perhaps because they are using different languages) for the same concepts so that the pairs {true, false}, {T,F}, {0,1}, {vrai, faux} could be different representations of the Boolean sort.

What is essential is the abstract relationship between identities and not the concrete representation. Thus for numerals what is interesting is the relationship between 1 and 2 which is the same as the relationship between I and II. Also for operators what is of interest is the relationship between the operator identity and other operator identities and the list of arguments. Concrete constructions such as brackets which allow us to distinguish between (a+b)*c and a+(b*c) are only of interest so that the underlying abstract concept can be determined.

These abstract concepts are embodied in an abstract syntax of the concept which may be realised by more than one concrete syntax. For example the following two concrete examples both describe the same data type properties but in different concrete syntax.

```
NEWTYPE     bool LITERALS true, false;
OPERATORS   "not" :bool ->bool;
AXIOMS
        not(true)       == false;
        not(not(a))     == a;
ENDNEWTYPE bool;

NEWTYPE int LITERALS zero, one;
OPERATORS   plus        :int,int ->int;
            minus       :int,int ->int;
AXIOMS
        plus(zero,a)                == a;
        plus(a,b)                   == plus(b,a);
        plus(a,plus(b,c))           == plus(plus(a,b),c);
        minus(a,a)                  == zero;
        minus(a,zero)               == a;
        minus(a,minus(b,c))         == minus(plus(a,c),b);
        minus(minus(a,b),c)         == minus(a,plus(b,c));
        plus(minus(a,b),c)          == minus(plus(a,c),b);
ENDNEWTYPE int;

NEWTYPE tree LITERALS nil;
OPERATORS
        tip         : int       ->tree;
        isnil       : tree      ->bool;
        istip       : tree      ->bool;
        node        : tree,tree ->tree;
        sum         : tree      ->int;
AXIOMS
        istip(nil)              == false;
        istip(tip(i))           == true;
        istip(node(t1,t2))      == false;
        isnil(nil)              == true;
        isnil(tip(i))           == false;
        isnil(node(t1,t2))      == false;
        sum(node(t1,t2))        == plus(sum(t1),sum(t2));
        sum(tip(i))             == i;
        sum(nil)                == zero;
ENDNEWTYPE tree;
```

EXAMPLE 1

```
TYPE      bool      IS
SORTS     bool
OPNS      true  :      -> bool
          false :      -> bool
          not   :bool-> bool
EQNS OFSORT      bool FOR ALL a:bool
      not(true)      = false;
      not(not(a))    = a
ENDTYPE

TYPE      int   IS   bool WITH
SORTS     int
OPNS      zero :                -> int
          one  :                -> int
          plus :   int,int      -> int
          minus:   int,int      -> int
EQNS OFSORT      int   FOR ALL a,b,c:int
      plus(zero,a)              = a ;
      plus(a,b)                 = plus(b,a);
      plus(a,plus(b,c))         = plus(plus(a,b),c) ;
      minus(a,a)                = zero;
      minus(a,zero)             = a ;
      minus(a,minus(b,c))       = minus(plus(a,c),b);
      minus(minus(a,b),c)       = minus(a,plus(b,c));
      plus(minus(a,b),c)        = minus(plus(a,c),b)
ENDTYPE


TYPE      tree  IS   int   WITH
SORTS     tree
OPNS      nil   :           ->tree
          tip   : int       ->tree
          isnil : tree      ->bool
          istip : tree      ->bool
          node: tree,tree   ->tree
          sum : tree        ->int
EQNS OFSORT      bool FOR ALL i:int, t1,t2:tree
          istip(nil)              = false;
          istip(tip(i))           = true ;
          istip(node(t1,t2))      = false;
          isnil(nil)              = true ;
          isnil(tip(i))           = false;
          isnil(node(t1,t2))      = false
      OFSORT      int   FOR ALL i:int, t1,t2:tree
          sum(node(t1,t2))        = plus(sum(t1),sum(t2));
          sum(tip(i))             = i ;
          sum(nil)                = zero
ENDTYPE
```

EXAMPLE 2

This example will be used for illustration. Initially the definition of sorts and literals will be considered.

It should be noted that literals are considered to be a special case of operators, that is operators without parameters.

We can introduce some sorts and literals in the first form by

```
NEWTYPE int LITERALS zero, one; ...
NEWTYPE bool LITERALS true, false; ...
NEWTYPE tree LITERALS nil; ...
```

or in the second form by

```
...
SORTS   bool
OPNS    true :      -> bool
        false :     -> bool
...
SORTS   int
OPNS    zero :      -> int
        one  :      -> int
...
SORTS   tree
OPNS    nil  :      ->tree
...
```

In the following the second form only will be used as that is closest to the formulation used in many publications on initial algebra. It should be noted that the form of terms is the same in both cases and the most significant difference is the way in which literals are introduced. It should be remembered that it is necessary to adopt a concrete notation to communicate the concepts, but the meaning of the algebras is independent of the notation so that systematic renaming of names (retaining the same uniqueness) and a change from prefix to polish notation will not change the meaning defined by the type definitions.

5.3.2 *Signatures*

Associated with each sort will be one or more operators. Each operator has an operator functionality; that is it is defined to relate one or more input sorts to a result sort.

For example the following operators can be added to the sorts defined above

```
...
SORTS   bool
OPNS    true :      -> bool
        false :     -> bool
        not  :bool  -> bool
...
SORTS   int
OPNS    zero :              -> int
        one  :              -> int
        plus :      int,int -> int
        minus:      int,int -> int
...
SORTS   tree
OPNS    nil  :              ->tree
        tip  :      int     ->tree
        isnil :     tree    ->bool
        istip :     tree    ->bool
        node:       tree,tree ->tree
        sum :       tree    ->int
...
```

The signature of the type which applies is the set of sorts, and the set of operators (both literals and operators with parameters) which are visible.

A signature of a type is called complete (closed) if for every operator in the signature, the sorts of the functionality of the operator are included in the set of sorts of the type.

5.3.3 *Terms and expressions*

The language of interest is one which allows expressions which are variables, literals or operators applied to expressions. A variable is a data object which is associated with an expression. Interpretation of a variable can be replaced with interpretation of the expression associated with the variable. In this way variables can be eliminated so that interpretation of an expression can be reduced to the application of various operators to literals.

Thus on interpretation an open expression (an expression involving variables) becomes a closed expression (an expression without variables) by providing the open expression with actual arguments (that is closed expressions).

A closed expression corresponds to a ground term.

The set of all possible ground terms of a sort is called the set of ground terms of the sort. For example for bool as defined above the set of ground terms will contain

{true, false, not(true), not(false), not(not(true)), ...}

It can be seen that even for this very simple sort the set of ground terms is infinite.

5.3.3.1 *Generation of terms*

Given a signature of a type it is possible to generate the set of ground terms for that type.

The set of literals of the type are considered to be the basic set of ground terms. Each literal has a sort, therefore each ground term has a sort. For the type being defined above this basic set of ground terms will be

{zero, one, true, false, nil}

For each operator in the set of operators for the type, ground terms are generated by substituting for each argument all previously generated ground terms of the correct sort for that argument. The result sort of each operator is the sort of the ground term generated by that operator. The resulting set of ground terms is added to the existing set of ground terms to generate a new set of ground terms. For the type above this is

{zero,	one,	true,	false,	nil,
plus(zero,zero),	plus(one,one),	plus(zero,one),	plus(one,zero),	
minus(zero,zero),	minus(one,one),	minus(zero,one),	minus(one,zero),	
not(true),	not(false),	tip(zero),	tip(one),	
isnil(nil),	istip(nil),	node(nil,nil),	sum(nil) }	

This new set of ground terms is then taken as the previous set of ground terms for a further application of the last algorithm to generate a further set of ground terms. This set of ground terms will include

```
{zero,              one,              true,                 false,              nil,
 plus(zero,zero),   plus(one,one),    plus(zero,one),       plus(one,zero),     ...
 plus(zero,plus(zero,zero)),          plus(zero,plus(one,one)),                 ...
 plus(zero,sum(nil)),                 ...
 isnil(node(nil,nil)),                istip(node(nil,nil)), node(nil,node(nil,nil)),
 ...,                                                       sum(node(nil,nil)) }
```

This algorithm is applied repeatedly to generate all possible ground terms for the type which is the set of ground terms for the type. The set of ground terms for a sort is the set of ground terms of the type which have that sort.

Normally generation will continue indefinitely yielding an infinite number of terms.

5.3.4 *Values and algebras*

Each term of a sort represents a value of that sort. It can be seen from above that even a simple sort such as bool has an infinite number of terms and hence an infinite number of values, unless some definition is given of how terms are equivalent (that is represent the same value). This definition is given by equations defined on terms. In the absence of istip and isnil the sort bool can be limited to two values by the equations

```
not(true)  = false;
not(false) = true
```

Such equations define terms to be equivalent and it is then possible to obtain the two equivalent classes of terms

```
{ true ,  not(false),   not(not(true )),   not(not(not(false))), ...}
{ false,  not(true ),   not(not(false)),   not(not(not(true ))), ...}
```

Each equivalence class then represents one value and members of the class are different representations of the same value.

Note that unless they are defined equivalent by equations, terms are non-equivalent (that is they do not represent the same value).

An algebra defines the set of terms which satisfies the signature of the algebra. The equations of the algebra relate terms to one another.

In general there will be more than one representation for each value of a sort in an algebra.

An algebra for a given signature is an initial algebra if and only if any other algebra which gives the same properties for the signature can be systematically transformed onto the initial algebra. (Formally such a transformation is known as a homomorphism.)

Providing not, istip and isnil always produce values in the equivalence classes of true and false then an initial algebra for bool is the pair of literals

```
{true, false}
```

and no equations.

5.3.4.1 *Equations and quantification*

For a sort such as bool, where there are only a limited number of values, all equations can be written using only ground terms, that is terms which only contain literals and operators.

When a sort contains many values, writing all the equations using ground terms is not practical and for sorts with an infinite number of values (such as integers), such explicit enumeration becomes impossible. The technique of writing quantified equations is used to represent a possibly infinite set of equations by one quantified equation.

A quantified equation contains value identifiers in terms. Such terms are called composite terms. The set of equations with only ground terms can be derived from the quantified equation by systematically generating equations with each value identifier substituted in the equation by one of the ground terms of the sort of the value identifier. For example

```
FOR ALL b : bool    not(not(b))=b
```

represents

```
not(not(true))  = true;
not(not(false)) = false
```

An alternative set of equations for bool can now be taken as

```
FOR ALL b : bool
    not(not(b))     = b ;
    not(true)       = false
```

When the sort of the quantified value identifier is obvious from context it is usual practice to omit the clause defining the value identifier so that the example becomes

```
not(not(b))     = b ;
not(true)       = false
```

5.3.5 *Algebraic specification and semantics (meaning)*

An algebraic specification consists of a signature and sets of equations for each sort of that signature. These sets of equations induce equivalence relations which define the meaning of the specification.

The symbol = denotes an equivalence relation that satisfies the reflexive, symmetric and transitive properties and the substitution property.

The equations given with a type allow terms to be placed into equivalence classes. Any two terms in the same equivalence class are interpreted as having the same value. This mechanism can be used to identify syntactically different terms which have the same intended value.

Two terms of the same sort, TERM1 and TERM2, are in the same equivalence class if

a) there is an equation
   TERM1=TERM2,
   or

b) one of the equations derived from the given set of quantified equations is
   TERM1=TERM2,
   or

c) i) TERM1 is in an equivalence class containing TERMA, and

ii) TERM2 is in an equivalence class containing TERMB, and

iii) there is an equation or an equation derived from the given set quantified equations such that
TERMA=TERMB,

or

d) by substituting a sub-term of TERM1 by a term of the same class as the sub-term producing a term TERM1A it is possible to show that TERM1A is in the same class as TERM2.

By applying all equations the terms of each sort are partitioned into one or more equivalence classes. There are as many values for the sort as there are equivalence classes. Each equivalence class represents one value and every member of a class represents the same value.

5.3.6 *Representation of values*

Interpretation of an expression then means first deriving the ground term by determining the actual value of variables used in the expression at the point of interpretation, then finding the equivalence class of this ground term. The equivalence class of this term determines the value of the expression.

Meaning is thus given to operators used in expressions by determining the resultant value given a set of arguments.

It is usual to choose a literal in the equivalence class to represent the value of the class. For instance bool would be represented by true and false, and natural numbers by 0,1,2,3 etc.. When there is no literal then usually a term of the lowest possible complexity (least number of operators) is used. For instance for negative integers the usual notation is -1, -2 -3 etc..

5.4 *Passive use of SDL data*

In § 5.4.1 extensions to the data definition constructs in § 5.2 are defined. How to interpret the use of the abstract data types in expressions is defined in § 5.4.2 if the expression is "passive" (that is do not depend on variables or the system state). How to interpret expressions which are not passive (that is "active" expressions) is defined in § 5.5.

5.4.1 *Extended data definition constructs*

The constructs defined in § 5.2 are the basis of more concise forms explained below.

Abstract grammar

There is no additional abstract syntax for most of these constructs. In § 5.4.1 and all subsections of § 5.4.1 the relevant abstract syntax is usually to be found in § 5.2.

Concrete textual grammar

<extended properties> ::=
    <inheritance rule>
  | <generator instantiations>
  | <structure definition>

<extended composite term> ::=
    <extended operator identifier> (<composite term list>)
  | <composite term> <infix operator> <term>
  | <term> <infix operator> <composite term>
  | <monadic operator> <composite term>
  | <conditional composite term>

<extended ground term> ::=
    <extended operator identifier>
        (<ground term> {, <ground term> }*)
  | <ground term> <infix operator> <ground term>
  | <monadic operator> <ground term>
  | <conditional ground term>

<extended operator identifier> ::=
    <operator identifier> <exclamation>
  | <generator formal name>
  | [<qualifier>] <quoted operator>

<extended operator name> ::=
    <operator name> <exclamation>
  | <generator formal name>
  | <quoted operator>

<exclamation> ::=
    !

<extended literal name> ::=
    <character string literal>
  | <generator formal name>
  | <name class literal>

<extended literal identifier> ::=
<character string literal identifier>
| <generator formal name>

The rules <extended properties>, <extended composite term>, <extended ground term>, <extended operator name>, <extended literal name> and <extended literal identifier> extend the rules for <partial type definition> (§ 5.2.1), <composite term> (§ 5.2.3), <ground term> (§ 5.2.3), <operator name> (§ 5.2.2), <literal> (§ 5.2.2) and <literal identifier> (§ 5.2.3) respectively in the data kernel. The rules above are further expanded by the rules <inheritance rule> (§ 5.4.1.11), <generator instantiations> (§ 5.4.1.12.2), <generator formal name> (§ 5.4.1.12.1), <conditional composite term> (§ 5.4.1.6), <conditional ground term> (§ 5.4.1.6), <character string literal> and <character string literal identifier> (§ 5.4.1.2) and <name class literal> (§ 5.4.1.14). The rules <infix operator>, <monadic operator>, <quoted infix operator> and <quoted monadic operator> are defined in § 5.4.1.1.

Alternatives with <generator formal name>s are only valid in a <properties expression> in a <generator text> (see § 5.4.1.12) which has that name defined as a formal parameter.

The alternatives of <extended composite term> and <extended ground term> with a <generator formal name> preceding a "(" are only valid if the <generator formal name> is defined to be of the OPERATOR class (see § 5.4.1.12).

The alternative of <extended literal name> with a <generator formal name> is only valid if the <generator formal name> is defined to be of the LITERAL class (see § 5.4.1.12).

The alternative of <extended literal identifier> with a <generator formal name> is only valid if the <generator formal name> is defined to be of the LITERAL class or the CONSTANT class (see § 5.4.1.12).

If an operator name is defined with an <exclamation>, then the <exclamation> is semantically part of the *name*.

The forms <operator name> <exclamation> or <operator identifier> <exclamation> represent *operator name* (§ 5.2.2) and *operator identifier* (§ 5.2.3) respectively.

Semantics

An operator name defined with an <exclamation> has the normal semantics of an operator, but the operator name is only visible in axioms.

5.4.1.1 *Special operators*

These are operator names which have special syntactic forms. The special syntax is introduced so that arithmetic operators and Boolean operators can have their usual syntactic form. That is the user can write "(1 + 1) = 2" rather than being forced to use the for example equal(add(1,1),2). Which sorts are valid for each operator will depend on the data type definition.

Concrete textual grammar

<quoted operator> ::=
<quote> <infix operator> <quote>
| <quote> <monadic operator> <quote>

<quote> ::=
"

<infix operator> ::=
        =>
    | OR
    | XOR
    | AND
    | IN
    | /=
    | =
    | >
    | <
    | <=
    | >=
    | +
    | /
    | *
    | //
    | MOD
    | REM
    | -

<monadic operator> ::=
        -
    | NOT

Semantics

An infix operator in a term has the normal semantics of an operator but with infix or quoted prefix syntax as above.

A monadic operator in a term has the normal semantics of an operator but with the prefix or quoted prefix syntax as above.

The quoted forms of infix or monadic operators are valid names for operators.

Infix operators have an order of precedence which determines the binding of operators. The binding is the same as the binding in <expression>s as specified in § 5.4.2.1.

When the binding is ambiguous such as in
        a OR b XOR c ;
then binding is from left to right so that the above term is equivalent to
        (a OR b) XOR c ;

Note that the <quoted operator>s MOD and REM have no predefined semantics, as they are not defined in the predefined data sorts.

Model

A term of the form
        <term1> <infix operator> <term2>
is derived syntax for
        "<infix operator>" (<term1>, <term2>)
with "<infix operator>" as a legal name. "<infix operator>" represents an *operator name.*

Similarly
        <monadic operator> <term>
is derived syntax for
        "<monadic operator>" (<term>)

with "<monadic operator>" as a legal name and representing an *operator name*.

(Note that the SDL equality operator (=) should not be confused with the SDL term equivalence symbol (==).)

5.4.1.2 *Character string literals*

Concrete textual grammar

<character string literal identifier> ::=
[<qualifier>] <character string literal>

<character string literal> ::=
<character string>

A <character string> is a lexical unit defined in § 2.2.1.

A <character string literal identifier> represents a *Literal-operator-identifier* in the abstract syntax.

A <character string literal> represents a unique *Literal-operator-name* (§ 5.2.2) in the abstract syntax derived from the <character string>.

Semantics

Character string literal identifiers are the identifiers formed from character string literals in terms and expressions.

Character string literals are used for the predefined data sorts Charstring and Character (see § 5.6). They also have a special relationship with name class literals (see § 5.4.1.14) and literal mappings (see § 5.4.1.15). These literals may also be defined to have other uses.

A <character string literal> has a length which is the number of <alphanumeric>s plus <other character>s plus <special>s plus <full stop>s plus <underline>s plus <space>s plus <apostrophe> <apostrophe> pairs in the <character string> (see § 2.2.1).

A <character string literal> which

a) has a length greater than one, and

b) has a substring formed by deleting the last character (<alphanumeric> or <other character> or <special> or <full stop> or <underline> or <space> or <apostrophe> <apostrophe> pairs) from the <character string>, and

c) that substring is defined as a literal such that
substring // deleted_character_in_quotes
is a valid term with the same sort as the <character string literal>,

then there is an implied equation given by the concrete syntax that the <character string literal> is equivalent to the substring followed by the "//" infix operator followed by the deleted character with apostrophes to form a <character string>.

For example the literals 'ABC', 'AB'', and 'AB' in

```
NEWTYPE s
LITERALS 'ABC', 'AB'', 'AB', 'A', 'B', '';
OPERATORS "//": s, s -> s;
```

have implied equations

```
'ABC'   == 'AB' // 'C' ;
'AB''   == 'AB' // '' ;
'AB'    == 'A'  // 'B';
```

5.4.1.3 *Predefined data*

The predefined data including the Boolean sort which defines properties for two literals True and False, are defined in § 5.6. The semantics of Equality (§ 5.4.1.4), Boolean axioms (§ 5.4.1.5), Conditional terms (§ 5.4.1.6), Ordering (§ 5.4.1.8), and Syntypes (§ 5.4.1.9) rely on the definition of the Boolean sort (§ 5.6.1). The semantics of Name Class Literals (if <regular interval>s are used – § 5.4.1.14) and Literal Mapping (§ 5.4.1.15) also rely on the definition of Character (§ 5.6.2) and Charstring (§ 5.6.4) respectively.

Predefined data is considered to be defined at system level.

5.4.1.4 *Equality*

Concrete textual grammar

Each sort name introduced in a <partial type definition> has an implied *operator signature* for both = and /=, and an implied *equation* set for these operators.

A <partial type definition> introducing a sort named S has implied *operator signature* pair equivalent to

```
"=" : S, S -> Boolean;
"/=": S, S -> Boolean;
```

where Boolean is the predefined Boolean sort.

A <partial type definition> introducing a sort named S has an implied *equation* set

```
FOR ALL a, b, c IN S (
    a = a                          == True;
    a = b                          == b = a;
    (( a=b ) AND ( b=c )) => a=c   == True;
    a /= b                         == NOT ( a=b );
    a = b == True  ==>      a      == b    )
```

The last equation expresses the substitution property for equality.

If it is possible to derive from the equations (explicit, implicit and derived) that

```
True == False
```

this is in contradiction with the assumed properties of the Boolean data type and so the definition must be invalid. It must not be possible to derive

```
True == False;
```

Every Boolean ground expression which is used outside data type definitions must be interpreted as either True or False. If it is not possible to reduce such an expression to True or False then the specification is incomplete and allows more than one interpretation of the data type.

Semantics

For every sort introduced by a partial data type definition there is an implicit definition of operators and equations for equality.

The symbols = and /= in the concrete syntax represent the names of the operators which are called the equal and not equal operators.

5.4.1.5 *Boolean axioms*

Concrete textual grammar

<Boolean axiom> ::=
<Boolean term>

Semantics

A Boolean axiom is a statement of truth which holds under all conditions for the data type being defined, and thus can be used to specify the behaviour of the data type.

Model

An axiom of the form
<Boolean term>;
is derived syntax for the concrete syntax equation
<Boolean term> == True;
which has the normal relationship of an equation with the abstract syntax.

5.4.1.6 *Conditional terms*

In the following the equation containing the conditional term is called a conditional term equation.

Abstract grammar

Conditional-composite-term	=	*Conditional-term*
Conditional-ground-term	=	*Conditional-term*
Conditional-term	::	*Condition* *Consequence* *Alternative*
Condition	=	*Term*
Consequence	=	*Term*
Alternative	=	*Term*

The sort of the *Condition* must be the predefined Boolean sort and the *Condition* must not be the *Error-term*. The *consequence* and the *alternative* must have the same sort.

A *conditional term* is a *conditional composite term* if and only if one or more of the *terms* in the *condition*, the *consequence* or *alternative* is a *composite term*.

A *conditional term* is a *conditional ground term* if and only if all the *terms* in the *condition*, the *consequence* or *alternative* are *ground terms.*

Concrete textual grammar

<conditional composite term> ::=
<conditional term>

<conditional ground term> ::=
<conditional term>

<conditional term> ::=
IF <condition> THEN <consequence> ELSE <alternative> FI

<condition> ::=
<Boolean term>

<consequence> ::=
<term>

<alternative> ::=
<term>

Semantics

A conditional term used in an equation is semantically equivalent to two sets of equations where all the quantified value identifiers in the Boolean term have been eliminated.

The set equations can be formed by simultaneously substituting throughout the conditional term equation each *value identifier* in the *condition* by each *ground term* of the appropriate sort. In this set of equations the *condition* will always have been replaced by a Boolean *ground term.* In the following this set of equations is referred to as the expanded ground set.

A conditional term equation is equivalent to the *equation* set which contains

a) for every *equation* in the expanded ground set for which the *condition* is equivalent to True, that *equation* from the expanded ground set with the *conditional term* replaced by the (ground) *consequence*, and

b) for every *equation* in the expanded ground set for which the *condition* is equivalent to False, that *equation* from the expanded ground set with the *conditional term* replaced by the (ground) *alternative.*

Note that in the special case of an equation of the form
ex1 == IF a THEN b ELSE c FI;
this is equivalent to the pair of conditional equations
a == True ==> ex1 == b;
a == False ==> ex1 == c;

Example

IF i = j * j THEN posroot(i) ELSE abs(j) FI == IF positive(j) THEN j ELSE -j FI;

Note – There are better ways of specifying these properties - this is only an example.

5.4.1.7 *Errors*

Errors are used to allow the properties of a data type to be fully defined even for cases when no specific meaning can be given to the result of an operator.

Abstract grammar

Error-term :: ()

An *error term* must not be used as a argument *term* for an *operator identifier* in a *composite term.*

An *error term* must not be used as part of a *restriction.*

It must not be possible to derive from *Equations* that a *literal operator identifier* is equal to *error term.*

Concrete textual grammar

<error term> ::=
ERROR <exclamation>

Semantics

A term may be an error so that it is possible to specify the circumstances under which an operator produces an error. If these circumstances arise during interpretation then the further behaviour of the system is undefined.

5.4.1.8 *Ordering*

Concrete textual grammar

<ordering> ::=
ORDERING

(<ordering> is referenced in § 5.2.2)

Semantics

The ordering keyword is a shorthand for explicitly specifying ordering operators and a set of ordering equations for a partial type definition.

Model

A <partial type definition> introducing a sort named S with the keyword ORDERING implies an *operator signature* set equivalent to the explicit definitions:

"<" : S,S -> Boolean;
">" : S,S -> Boolean;
"<=": S,S -> Boolean;
">=": S,S -> Boolean;

where Boolean is the predefined Boolean sort, and also implies the Boolean *axioms*:

```
FOR ALL a,b IN S
(
        "<"(a,a)   == False;
        "<"(a,b)   == ">"(b,a);
        "<="(a,b) == "OR"("<"(a,b),"="(a,b));
        ">="(a,b) == "OR"(">"(a,b),"="(a,b));
        "<"(a,b)   => NOT("<"(b,a)) ;
        "<"(a,b)   AND "<"(b,c) => "<"(a,c) ;
);
```

When a <partial type definition> includes both <literal list> and the keyword ORDERING the <literal signature>s are nominated in ascending order, that is

```
        LITERALS A,B,C;
        OPERATORS ORDERING;
```

implies A<B, B<C.

5.4.1.9 *Syntypes*

A syntype specifies set of values of a sort. A syntype used as a sort has the same semantics as the sort referenced by the syntype except for checks that values are within the value set of the sort.

Abstract grammar

Syntype-identifier	=	*Identifier*
Syntype-definition	::	*Syntype-name* *Parent-sort-identifier* *Range-condition*
Syntype-name	=	*Name*
Parent-sort-identfier	=	*Sort-identifier*

Concrete textual grammar

```
<syntype> ::=
        <syntype identifier>

<syntype definition> ::=
        SYNTYPE
            <syntype name> = <parent sort identifier>
            [ <default assignment> ] [ CONSTANTS <range condition> ]
            ENDSYNTYPE [ <syntype name> ]
    |   NEWTYPE <syntype name> [ <extended properties> ]
            <properties expression> CONSTANTS <range condition>
            ENDNEWTYPE [ <syntype name> ]

<parent sort identifier> ::=
        <sort>
```

A <syntype> is an alternative for a <sort> (see § 5.2.2).

A <syntype definition> with the keyword SYNTYPE and "= <syntype identifier>" is derived syntax defined below.

A <syntype definition> with the keyword SYNTYPE in the concrete syntax corresponds to a *Syntype-definition* in the abstract syntax.

A <syntype definition> with the keyword NEWTYPE can be distinguished from a <partial type definition> by the inclusion of CONSTANTS <range condition>. Such a <syntype definition> is a shorthand for introducing a <partial type definition> with an anonymous name followed by a <syntype definition> with the keyword SYNTYPE based on this anonymously named sort. That is

```
NEWTYPE X /* details */
    CONSTANTS /* constant list */
ENDNEWTYPE X;
```

is equivalent to

```
NEWTYPE anon /* details */
ENDNEWTYPE anon;
```

followed by

```
SYNTYPE X = anon
    CONSTANTS /* constant list */
ENDSYNTYPE X;
```

When a <syntype identifier> is used as an <argument> in an <argument list> defining an operator, the sort for the argument in an *argument list* is the *parent sort identifier* of the syntype.

When a <syntype identifier> is used as a result of an operator, the sort of the *result* is the *parent sort identifier* of the syntype.

When a <syntype identifier> is used as a qualifier for a name, the *qualifier* is the *parent sort identifier* of the syntype.

The optional <syntype name> given at the end of a <syntype definition> after the keyword ENDSYNTYPE or ENDNEWTYPE must be the same as the <syntype name> specified after SYNTYPE or NEWTYPE respectively.

If the keyword SYNTYPE is used and the <range condition> is omitted then all the values of the sort are in the range condition so that the <syntype identifier> has exactly the same semantics as the sort identifier and the range condition is always true.

Semantics

A syntype definition defines a syntype which references a sort identifier and range condition. Specifying a syntype identifier is the same as specifying the parent sort identifier of the syntype except for the following cases:

a) assignment to a variable declared with a syntype (see § 5.5.3),

b) an output of a signal if one of the sorts specified for the signal is a syntype (see § 2.7.4),

c) calling a procedure when one of the sorts specified for the procedure IN parameter variables is a syntype (see § 2.4.5),

d) creating a process when one of the sorts specified for the process parameters is a syntype (see § 2.7.2 and § 2.4.4),

e) input of a signal and one of the variables which is associated with the input, has a sort which is a syntype (see § 2.6.4),

f) use in an expression of an operator which has a syntype defined as either an argument sort or a result sort (see § 5.4.2.2 and § 5.5.2.4),

g) a set or reset statement on a timer and one of the sorts in the timer definition is a syntype (see § 2.8),

h) an import definition (see §4.13).

For example a <syntype definition> with the keyword SYNTYPE and "= <syntype identifier>" is equivalent to substituting the <parent sort identifier> by the <parent sort identifier> of the <syntype definition> of the <syntype identifier>. That is

```
SYNTYPE s2 = n1 CONSTANTS a1:a3; ENDSYNTYPE s2;
SYNTYPE s3 = s2 CONSTANTS a1:a2; ENDSYNTYPE s3;
```

is equivalent to

```
SYNTYPE s2 = n1 CONSTANTS a1:a3; ENDSYNTYPE s2;
SYNTYPE s3 = n1 CONSTANTS a1:a2; ENDSYNTYPE s3;
```

When a syntype is specified in terms of <syntype identifier> then the two syntypes must not be mutually defined.

A syntype defined by a syntype definition has an identity which is the name introduced by the syntype name qualified by the identity of the enclosing scope unit.

A syntype has a sort which is the sort identified by the parent sort identifier given in the syntype definition.

A syntype has a range which is the set of values specified by the constants of the syntype definition.

5.4.1.9.1 *Range condition*

Abstract grammar

Range-condition	::	*Or-operator-identifier* *Condition-item*–**set**
Condition-item	=	*Open-range* | *Closed-range*
Open-range	::	*Operator-identifier* *Ground-expression*
Closed-range	::	*And-operator-identifier* *Open-range* *Open-range*
Or-operator-identifier	=	*Identifier*
And-operator-identifier	=	*Identifier*

Concrete textual grammar

<range condition> ::=
{ <closed range> | <open range> } { , { <closed range> | <open range> } }*
<closed range> ::=
<constant> : <constant>

<open range> ::=
<constant>
| { = | /= | < | > | <= | >= } <constant>

<constant> ::=
<ground expression>

The symbol "<" ("<=", ">", ">=" respectively) must only be used in the concrete syntax of the <range condition> if that symbol has been defined with an <operator signature>
P, P -> Boolean;
where P is the sort of the syntype. These symbols represent *operator identifier*.

A <closed range> must only be used if the symbol "<=" is defined with an <operator signature>
P, P -> Boolean;
where P is the sort of the syntype.

A <constant> in a <range condition> must have the same sort as the sort of the syntype.

Semantics

A range condition defines a range check. A range check is used when a syntype has additional semantics to the sort of the syntype (see § 5.4.1.9 and the cases where syntypes have different semantics – see § 5.5.3, § 2.6.4, § 2.7.2, § 2.5.4, § 5.4.2.2 and § 5.5.4). A range check is also used to determine the interpretation of a decision (see § 2.7.5).

The range check is the application of the operator formed from the range condition. The application of this operator must be equivalent to true otherwise the further behaviour of the system is undefined. The range check is derived as follows:

a) Each element (<open range> or <closed range>) in the <range condition> has a corresponding *open range* or *closed range* in the *condition item.*

b) An <open range> of the form <constant> is equivalent to an <open range> of the form = <constant>.

c) For a given term, A, then

i) an <open range> of the form = <constant>, /= <constant>, < <constant>, <= <constant>, > <constant>, and >= <constant>, has sub-terms in the range check of the form A = <constant>, A /= <constant>, A < <constant>, A <= <constant>, A > <constant>, and A >= <constant> respectively.

ii) a <closed range> of the form <first constant> : <second constant> has a sub-term in the range check of the form <first constant> <= A AND A <= <second constant> where AND corresponds to the Boolean AND operator and corresponds to the *And operator identifier* in the abstract syntax.

d) There is an *or operator identifier* for the distributed operator over all the elements in the

condition-item–**set** which is a Boolean union (OR) of all the elements. The range check is the term formed from the Boolean union (OR) of all the sub-terms derived from the <range condition>.

If a syntype is specified without a <range condition> then the range check is True.

5.4.1.10 *Structure sorts*

Concrete textual grammar

<structure definition> ::=
STRUCT <field list> [<end>] [ADDING]

<field list> ::=
<fields> { <end> <fields> }*

<fields> ::=
<field name> { , <field name> }* <field sort>

<field sort> ::=
<sort>

Each <field name> of a structure sort must be different from every other <field name> of the same <structure definition>.

Semantics

A structure definition defines a structure sort whose values are composed from a list of field values of sorts.

The length of the list of values is determined by the structure definition and the sort of a value is determined by its position in the list of values.

Model

A structure definition is derived syntax for the definition of

a) an operator, Make!, to create structure values, and

b) operators both to modify structure values and to extract field values from structure values.

The name of the implied operator for modifying a field is the field name concatenated with "Modify!".

The name of the implied operator for extracting a field is the field name concatenated with "Extract!".

The <argument list> for the Make! operator is the list of <field sort>s occurring in the field list in the order in which they occur.

The <result> for the Make! operator is the sort identifier of the structure.

The <argument list> for the field modify operator is the sort identifier of the structure followed by the <field sort> of that field. The <result> for a field modify operator is the sort identifier of the structure.

The <argument list> for a field extract operator is the sort identifier of the structure. The <result> for a field extract operator is the <field sort> of that field.

There is an implied equation for each field which defines that modifying a field of a structure to a value is the same as constructing a structure value with that value for the field.

There is an implied equation for each field which defines that extracting a field of a structure value will return the value associated with that field when the structure value was constructed.

For example

```
NEWTYPE s STRUCT
    b   Boolean;
    i   Integer;
    c   Character;
ENDNEWTYPE   s;
```

implies

```
NEWTYPE s
OPERATORS
    Make!       : Boolean, Integer, Character   -> s;
    bModify!    : s, Boolean                    -> s;
    iModify!    : s, Integer                    -> s;
    cModify!    : s, Character                  -> s;
    bExtract!   : s                             -> Boolean;
    iExtract!   : s                             -> Integer;
    cExtract!   : s                             -> Character;
AXIOMS
    bModify!    (Make!(x,y,z),b)                == Make!(b,y,z);
    iModify!    (Make!(x,y,z),i)                == Make!(x,i,z);
    cModify!    (Make!(x,y,z),c)                == Make!(x,y,c);
    bExtract!   (Make!(x,y,z))                  == x;
    iExtract!   (Make!(x,y,z))                  == y;
    cExtract!   (Make!(x,y,z))                  == z;
ENDNEWTYPE s;
```

5.4.1.11 *Inheritance*

Concrete textual grammar

<inheritance rule> ::=
    INHERITS <parent sort> [<literal renaming>]
        [[OPERATORS] { ALL | (<inheritance list>) } [<end>]] [ADDING]

<parent sort> ::=
    <sort>

<inheritance list> ::=
    <inherited operator> { , <inherited operator> }*

<inherited operator> ::=
    [<operator name> =] <inherited operator name>

<inherited operator name> ::=
    <parent sort operator name>

<literal renaming> ::=
LITERALS <literal rename list> <end>

<literal rename list> ::=
<literal rename pair> { , <literal rename pair> }*
<literal rename pair> ::=
<literal rename signature> = < parent literal rename signature>

<literal rename signature> ::=
< literal operator name>
| <character string literal>

A sort must not be circularly based on itself by inheritance.

All <literal rename signature>s in a <literal rename list> must be distinct. All the <parent literal rename signature>s in a <literal rename list> must be different.

All <inherited operator name>s in an <inheritance list> must be distinct. All <operator name>s in an <inheritance list> must be distinct.

An <inherited operator name> specified in an <inheritance list> must be a visible operator of the <parent sort> defined in the <partial type definition> defining the <parent sort>. An operator name is not visible at this point if it is defined with an <exclamation>.

When several operators of the <parent sort> have the same name, as the <inherited operator name>, then all of these operators are inherited.

Semantics

One sort may be based on another sort by using NEWTYPE in combination with an inheritance rule. The sort defined using the inheritance rule is disjoint from the parent sort.

If the parent sort has literals defined the literal names are inherited as names for literals of the sort unless literal renaming has taken place for that literal. Literal renaming has taken place for a literal if the parent literal name appears as the second name in a literal renaming pair in which case the literal is renamed to the first name in that pair.

There is an inherited operator for every operator of the parent sort except "=" and "/=". An operator of the parent sort is any operator which both

a) is defined by any partial type definition or syntype definition (except that being defined) which defines a sort visible at the point of inheritance, and also

b) has the parent sort as either an argument or as a result.

The names of operators are inherited as specified by ALL or the inheritance list. The name of an inherited operator is

a) the same as the parent sort operator name if ALL is specified and the name is explicitly or implicitly defined as an operator name in the partial type definition or syntype definition defining the parent sort, otherwise

b) if the parent operator identifier is given in the inheritance list and an operator name followed by "=" is given for the inherited operator, then renamed to this name, otherwise

c) if the parent operator identifier is given in the inheritance list and an operator name followed by "=" is not given for the inherited operator, then the same name as the parent sort operator name, otherwise

d) if ALL is not specified and the parent operator identifier is not mentioned in the inheritance list, then renamed to an invisible but unique name. Such names cannot be explicitly used either in axioms or expressions.

The argument sorts and result of an inherited operator are the same as those of the corresponding operator of the parent sort, except if the argument sort or result is the parent sort in which case it is changed to the sort being defined. That is every occurrence of the parent sort in the inherited operators is changed to the new sort.

From each equation of the parent sort an equation is derived by inheritance. The equations of the parent sort are

a) any equation which contains an operator (or literal) of the parent sort, and also

b) any equation which is defined by any partial type definition or syntype definition (except that being defined) which defines a sort visible at the point of inheritance.

An inherited equation is the same as the corresponding equation of the parent sort except that

a) any occurrence of the parent sort is changed to the new sort, and

b) operators (or literals) of the parent sort which have renamed inherited operators (or literals), undergo the same renaming in the inherited equation.

As a consequence of changing sorts as in (a) the literal identities and operator identities of inherited literals and inherited operators are changed to be qualified by the sort identity of the new sort.

Model

The concrete syntax of an <inheritance rule> is related to the concrete syntax of the <properties expression> in the <partial type definition> or <syntype definition> containing the <inheritance rule>.

The set of <literal>s of the new sort in the abstract syntax corresponds to the set of <literal signature>s in the <properties expression> plus the set of inherited literals.

The set of <operator>s of the new sort in the abstract syntax corresponds to the set of <operator signature>s in the <properties expression> plus the set of inherited operators.

The set of <equations> of the new sort in the abstract syntax corresponds to the <axioms> of the <properties expression> plus the set of inherited equations.

Example

```
NEWTYPE     bit
        INHERITS    Boolean
            LITERALS     1 = True, 0 = False;
            OPERATORS  ("NOT", "AND", "OR")
        ADDING
        OPERATORS
            EXOR: bit,bit -> bit;
        AXIOMS /* note – 2 different ways of writing NOT are used here */
            EXOR(a,b) == (a AND "NOT"(b)) OR (NOT a AND b );
ENDNEWTYPE bit;
```

5.4.1.12 *Generators*

A generator allows a parameterised text template to be defined which is expanded by instantiation before the semantics of data types are considered.

5.4.1.12.1 *Generator definition*

Concrete textual grammar

<generator definition> ::=
GENERATOR <generator name> (<generator parameter list>) <generator text>
ENDGENERATOR [<generator name>]

<generator text> ::=
[<generator instantiations>] <properties expression>

<generator parameter list> ::=
<generator parameter> { , <generator parameter> }*

<generator parameter> ::=
{ TYPE | LITERAL | OPERATOR | CONSTANT }
<generator formal name> {, <generator formal name> }*

<generator formal name> ::=
<generator formal name>

<generator sort> ::=
<generator formal name>
| <generator name>

A <generator name> or <generator formal name> must only be used in a <properties expression> if the <properties expression> is in a <generator text>.

In a <generator definition> all <generator formal name>s of the same class (TYPE, LITERAL, OPERATOR or CONSTANT) must be distinct. A name of the class LITERAL must be distinct from every name of the class CONSTANT in the same <generator definition>.
The <generator name> after the keyword GENERATOR must be distinct from all sort names in the <generator definition> and also distinct from all TYPE <generator parameter>s of that <generator definition>.

A <generator sort> is only valid if it appears as an <extended sort> (see § 5.2.2) in a <generator text> and the name is either the <generator name> of that <generator definition> or a <generator formal name> defined by that definition.

If a <generator sort> is a <generator formal name> it must be a name defined to be of the TYPE class.

The optional <generator name> after ENDGENERATOR must be the same as the <generator name> given after GENERATOR.

A <generator formal name> must not be used in a <qualifier>. A <generator name> or <generator formal name> must not:

a) be qualified, or
b) be followed by an <exclamation>, or
c) be used in a <default assignment>.

Semantics

A generator names a piece of text which can be used in generator instantiations.

The texts of generator instantiations within a generator text are considered to be expanded at the point of definition of the generator text.

Each generator parameter has a class (TYPE, LITERAL, OPERATOR or CONSTANT) specified by the keyword TYPE, LITERAL, OPERATOR or CONSTANT respectively.

Model

The text defined by a generator definition is only related to the abstract syntax if the generator is instantiated. There is no corresponding abstract syntax for the generator definition at the point of definition.

Example

```
GENERATOR bag(TYPE item)
LITERALS empty;
OPERATORS
        put     : item, bag -> bag;
        count   : item, bag -> Integer;
        take    : item, bag -> bag;
AXIOMS
        take(i,put(i,b))        == b;
        take(i,empty)           == ERROR!;
        count(i,empty)          == 0;
        count(i,put(j,b))       == count(i,b) + IF i=j THEN 1 ELSE 0 FI;
        put(i,put(j,b))         == put(j,put(i,b));
ENDGENERATOR bag;
```

Note – The formal definition (Annex F.2) does not allow the use of <generator formal name> in qualifiers. The recommendation was corrected for this topic, after the Annex F.2 was printed. Annex F.2 is thus invalid on this topic.

5.4.1.12.2 *Generator instantiation*

Concrete textual grammar

```
<generator instantiations> ::=
        { <generator instantiation> [ <end> ] [ ADDING ] }+

<generator instantiation> ::=
        <generator identifier> ( <generator actual list> )

<generator actual list> ::=
        <generator actual> { , <generator actual> }*

<generator actual> ::=
        <extended sort>
    |   <literal signature>
    |   <operator name>
    |   <ground term>
```

If the class of a <generator parameter> is TYPE then the corresponding <generator actual> must be an <extended sort>.

If the class of a <generator parameter> is LITERAL then the corresponding <generator actual> must be a <literal signature>.
A <literal signature> which is a <name class literal> may be used as a <generator actual> if and only if the corresponding <generator formal name> does not occur in the <axioms>, or <literal mapping> of the <properties expression> in the <generator text>.

If the class of a <generator parameter> is OPERATOR then the corresponding <generator actual> must be an <operator name>.

If the class of a <generator parameter> is CONSTANT then the corresponding <generator actual> must be a <ground term>.

If the <generator actual> is a <generator formal name> then the class of the <generator formal name> must be the same as the class for the <generator actual>.

Semantics

Use of a generator instantiation in extended properties or in a generator text denotes instantiation of the text identified by the generator identifier. An instantiated text for literals, operators and axioms is formed from the generator text with

a) the generator actual parameters substituted for the generator parameters, also

b) with the name of the generator substituted by

   i) if the generator instantiation is in a partial type definition or syntype definition, the identity of the sort being defined by the partial type definition or syntype definition, otherwise

   ii) in the case of generator instantiation within a generator, the name of that generator.

The instantiated text for literals is the text instantiated from the literals in the properties expression of the generator text omitting the keyword LITERALS.

The instantiated text for operators is the text instantiated from the operator list in the properties expression of the generator text omitting the keyword OPERATORS.

The instantiated text for axioms is the text instantiated from the axioms in the properties expression of the generator text omitting the keyword AXIOMS.

When there is more than one generator instantiation in the list of generator instantiations, the instantiated texts for literals (operators and axioms) are formed by concatenating the instantiated text for the literals (operators, axioms respectively) of all the generators in the order they appear in the list.

The instantiated text for literals is a list of literals for the properties expression of the enclosing partial type definition, syntype definition or generator definition occurring before any literal list explicitly mentioned in the properties expression. That is if ordering has been specified, literals defined by generator instantiations will be in the order they are instantiated and before any other literals.

The instantiated text for operators and axioms are added to the operator list and axioms respectively of the enclosing partial type definition, syntype definition or generator definition.

When instantiated text is added to a properties expression the keywords LITERALS, OPERATORS and AXIOMS are considered to be added if necessary to create correct concrete syntax.

Model

The abstract syntax corresponding to a generator instantiation is determined after instantiation. The relationship is determined from the instantiated text at the point of instantiation.

Example

```
NEWTYPE boolbag bag(Boolean)
        ADDING
        OPERATORS
            yesvote : boolbag -> Boolean;
        AXIOMS
            yesvote(b) == count(True,b) > count(False,b);
ENDNEWTYPE boolbag;
```

5.4.1.13 *Synonyms*

A synonym gives a name to a ground expression which represents one of the values of a sort.

Concrete textual grammar

```
<synonym definition> ::=
            SYNONYM <synonym name> [ <sort> ] = <ground expression>
        |   <external synonym definition>
```

The alternative <external synonym definition> is described in § 4.3.1.

If the sort of the <ground expression> cannot be uniquely determined, then a sort must be specified in the <synonym definition>.

The sort identified by the <sort> must be one of the sorts to which the <ground expression> can be bound.

The <ground expression> must not refer to the synonym defined by the <synonym definition> either directly or indirectly (via another synonym).

Semantics

The value which the synonym represents is determined by the context in which the synonym definition appears.

If the sort of the ground expression cannot be uniquely determined in the context of the synonym then the sort is given by the <sort>.

A synonym has a value which is the value of the ground term in the synonym definition.

A synonym has a sort which is the sort of the ground term in the synonym definition.

Model

The <ground expression> in the concrete syntax denotes a *ground term* in the abstract syntax as

defined in § 5.4.2.2.

If a <sort> is specified the result of the <ground expression> is bound to that *sort*. The <ground expression> represents a *ground term* in the abstract syntax which has an *operator identifier* with the same name and the same argument sorts as given by the concrete syntax and the result sort equal to the sort specified in the concrete syntax.

5.4.1.14 *Name class literals*

A name class literal is a shorthand for writing a (possibly infinite) set of literal names defined by a regular expression.

Concrete textual grammar

<name class literal> ::=
NAMECLASS <regular expression>

<regular expression> ::=
<partial regular expression>
{ [OR] <partial regular expression> }*

<partial regular expression>::=
<regular element> [<natural literal name> | + | *]

<regular element> ::=
(<regular expression>)
| <character string literal>
| <regular interval>

<regular interval> ::=
<character string literal> : <character string literal>

The names formed by the <name class literal> must satisfy the normal static conditions for literals (see § 5.2.2) and either the lexical rules for names (see § 2.2.1) or the concrete syntax for <character string literal> (see § 5.4.1.2).

The <character string literal>s in a <regular interval> must both be of length one, and must both be literals defined by the Character sort (see § 5.6.2).

Semantics

A name class literal is an alternative way of specifying literal signatures.

Model

The set of names which a name class literal is equivalent to is defined as the set of names which satisfy the syntax specified by the <regular expression>. The name class literal is equivalent to this set of names in the abstract syntax.

A <regular expression> which is a list of <partial regular expression>s without an OR specifies that the names can be formed from the characters defined by the first <partial regular expression> followed by the characters defined by the second <partial regular expression>.

When an OR is specified between two <partial regular expression>s then the names are formed from either the first or the second of these <partial regular expression>s. Note that OR is more

tightly binding than simple sequencing so that

NAMECLASS 'A' '0' OR '1' '2';

is equivalent to

NAMECLASS 'A' ('0' OR '1') '2';

and defines the literals A02, A12.

If a <regular element> is followed by <natural literal name> the <partial regular expression> is equivalent to the <regular element> being repeated the number of times specified by the <natural literal name>.

For example

NAMECLASS 'A' ('A' OR 'B') 2

defines names AAA, AAB, ABA and ABB.

If a <regular element> is followed an '*' the <partial regular expression> is equivalent to the <regular element> being repeated zero or more times.

For example

NAMECLASS 'A' ('A' OR 'B')*

defines names A, AA, AB, AAA, AAB, ABA, ABB, AAAA, ... etc.

If a <regular element> is followed an '+' the <partial regular expression> is equivalent to the <regular element> being repeated one or more times.

For example

NAMECLASS 'A' ('A' OR 'B')+

defines names AA, AB, AAA, AAB, ABA, ABB, AAAA, ... etc.

A <regular element> which is a bracketed <regular expression> defines the character sequences defined by the <regular expression>.

A <regular element> which <character string literal> defines the character sequence given in the character string literal (omitting the quotes).

A <regular element> which is a <regular interval> defines all the characters specified by the <regular interval> as alternative character sequences. The characters defined by the <regular interval> are all the characters greater than or equal to the first character and less than or equal to the second character according to the definition of the character sort (see § 5.6.2). For example

'a':'f'

defines the alternatives 'a' or 'b' or 'c' or 'd' or 'e' or 'f'.

If the sequence of definition of the names is important (for instance if ORDERING is specified), then the names are considered to be defined in the order so that they are alphabetically sorted according to the ordering of the character string sort. If two names commence with the same characters but are of different lengths then the shorter name is considered to be defined first.

5.4.1.15 *Literal mapping*

Literal mappings are shorthands used to define the mapping of literals to values.

Concrete Textual Grammar

<literal mapping> ::=

MAP <literal equation> { <end> <literal equation> }* [<end>]

<literal equation> ::=
<literal quantification>
(<literal axioms> { <end> <literal axioms> }* [<end>])

<literal axioms> ::=
<equation>
| <literal equation>

<literal quantification> ::=
FOR ALL <value name> { , <value name> }* IN <extended sort> LITERALS

<spelling term> ::=
SPELLING (<value identifier>)

The rules <literal mapping> and <spelling term> are not part of the data kernel but occur in the rules <properties expression> and <ground term> in § 5.2.1 and § 5.2.3 respectively.

Semantics

Literal mapping is a shorthand for defining a large (possibly infinite) number of axioms ranging over all the literals of a sort. The literal mapping allows the literals for a sort to be mapped onto the values of the sort. When the sort contains a large (or infinite) number of values a literal mapping is the only practical way to define the value corresponding to each literal.

The spelling term mechanism is used in literal mappings to refer to the character string which contains the spelling of the literal. This mechanism allows the Charstring operators to be used to define literal mappings.

Model

A <literal mapping> is a shorthand for a set of <axioms>. This set of <axioms> is derived from the <literal equation>s in the <literal mapping>. The <equation>s which are used for this derivation are all <equation>s contained in <axioms> of the rules <literal axioms>. In each of these <equation>s the <value identifier>s defined by the <value name> in the <literal quantification> are replaced. In each derived <equation> each occurrence of the same <value identifier> is replaced by the same <literal operator identifier> of the <sort> of the <literal quantification>. The derived set of <axioms> contains all possible <equation>s which can be derived in this way.
The derived <axioms> for <literal equation>s are added to <axioms> (if any) defined after the keyword AXIOMS and before the keyword MAP in the same <partial type definition>.

For example

```
NEWTYPE abc LITERALS 'A',b,'c';
    OPERATORS
        "<" : abc,abc -> Boolean;
        "+" : abc,abc -> Boolean;
    MAP FOR ALL x,y IN abc LITERALS
        (x < y => y + x);
    ENDNEWTYPE abc;
```

is derived concrete syntax for

```
NEWTYPE abc LITERALS 'A',b,'c';
    OPERATORS
        "<" : abc,abc -> Boolean;
        "+" : abc,abc -> Boolean;
```

```
AXIOMS
    'A'  < 'A'    => 'A'   + 'A';
    'A'  < b      => b     + 'A';
    'A'  < 'c'    => 'c'   + 'A';
     b   < 'A'    => 'A'   + b ;
     b   < b      => b     + b ;
     b   < 'c'    => 'c'   + b ;
    'c'  < 'A'    => 'A'   + 'c';
    'c'  < b      => b     + 'c';
    'c'  < 'c'    => 'c'   + 'c';
ENDNEWTYPE abc;
```

If a <literal quantification> contains one or more <spelling term>s then there is replacement of the <spelling term>s with Charstring literals (see § 5.6.3).

If the <literal signature> of the <literal operator identifier> of a <spelling term> is a <literal operator name> (see § 5.2.2), then the <spelling term> is shorthand for an uppercase Charstring derived from the <literal operator identifier>. The Charstring contains the uppercase spelling of the <literal operator name> of the <literal operator identifier>.

If the <literal signature> of the <literal operator identifier> of a <spelling term> is a <character string literal> (see § 5.2.2 and § 5.4.1.2), then the <spelling term> is shorthand for a Charstring derived from the <character string literal>. The Charstring contains the spelling of the <character string literal>.

The Charstring is used to replace the <value identifier> after the <literal equation> containing the <spelling term> is expanded as above.

For example

```
NEWTYPE abc LITERALS 'A',Bb,'c';
    OPERATORS
        "<" : abc,abc -> Boolean;
    MAP FOR ALL x,y IN abc LITERALS
        SPELLING(x) < SPELLING(y) => x < y;
    ENDNEWTYPE abc;
```

is derived concrete syntax for

```
NEWTYPE abc LITERALS 'A',Bb,'c';
    OPERATORS
        "<" : abc,abc -> Boolean;
    AXIOMS
        /* note that 'A', Bb, 'c' are bound to the local sort abc */
        /* '"A"', 'BB' and '"c"' should be qualified by the Charstring identifier
             if these literals are ambiguous - to be concise this is omitted below*/
        '"A"'   < '"A"'  => 'A'   < 'A';
        '"A"'   < 'BB'   => 'A'   < Bb;
        '"A"'   < '"c"'  => 'A'   < 'c';
        'BB'    < '"A"'  => Bb    < 'A';
        'BB'    < 'BB'   => Bb    < Bb;
        'BB'    < '"c"'  => Bb    < 'c';
        '"c"'   < '"A"'  => 'c'   < 'A';
        '"c"'   < 'BB'   => 'c'   < Bb;
        '"c"'   < '"c"'  => 'c'   < 'c';
    ENDNEWTYPE abc;
```

Every <unquantified equation> in <literal axioms> must contain a <spelling term> or a <literal operator identifier>.
A <spelling term> must be in a <literal mapping>.

The <value identifier> in a <spelling term> must be a <value identifier> defined by a <literal quantification>.

5.4.2 *Use of data*

The following defines how data types, sorts, literals, operators and synonyms are interpreted in expressions.

5.4.2.1 *Expressions*

Expressions are literals, operators, variables accesses, conditional expressions and imperative operators.

Abstract grammar

Expression = *Ground-expression* | *Active-expression*

An *expression* is an *active expression* if it contains an *active primary* (see § 5.5).

An *expression* which does not contain an *active primary* is a *ground expression.*

Concrete textual grammar

For simplicity of description no distinction is made between the concrete syntax of *ground expression* and *active expression.* The concrete syntax for <expression> is given in § 5.4.2.2 below.

Semantics

An expression is interpreted as the value of the ground expression or active expression. If the value is an error then the further behaviour of the system is undefined.

The expression has the sort of the ground expression or active expression.

5.4.2.2 *Ground expressions*

Abstract grammar

Ground-expression :: *Ground-term*

The static conditions for the *ground term* also apply to the *ground expression* .

Concrete textual grammar

<ground expression> ::=
<ground expression>

```
<expression> ::=
        <operand0>
    |   <sub expression> => <operand0>

<sub expression> ::=
        <expression>

<operand0> ::=
        <operand1>
    |   <sub operand0> { OR | XOR } <operand1>

<sub operand0> ::=
        <operand0>

<operand1> ::=
        <operand2>
    |   <sub operand1> AND <operand2>

<sub operand1> ::=
        <operand1>

<operand2> ::=
        <operand3>
    |   <sub operand2> { = | /= | > | >= | < | <= | IN } <operand3>

<sub operand2> ::=
        <operand2>

<operand3> ::=
        <operand4>
    |   <sub operand3> { + | - | // } <operand4>

<sub operand3> ::=
        <operand3>

<operand4> ::=
        <operand5>
    |   <sub operand4> { * | / | MOD | REM } <operand5>

<sub operand4> ::=
        <operand4>

<operand5> ::=
        [ - | NOT ] <primary>

<primary> ::=
        <ground primary>
    |   <active primary>
    |   <extended primary>

<ground primary> ::=
        <literal identifier>
    |   <operator identifier> ( <ground expression list> )
    |   ( <ground expression> )
    |   <conditional ground expression>
```

<extended primary> ::=
<synonym>
| <indexed primary>
| <field primary>
| <structure primary>

<ground expression list> ::=
<ground expression> { , <ground expression> }*

<operator identifier> ::=
<operator identifier>
| [<qualifier>] <quoted operator>

An <expression> which does not contain any <active primary> represents a *ground expression* in the abstract syntax. A <ground expression> must not contain an <active primary>.

If an <expression> is a <ground primary> with an <operator identifier> and an <argument sort> of the <operator signature> is a <syntype> then the range check for that syntype defined in § 5.4.1.9.1 is applied to the corresponding argument value. The value of the range check must be True.

If an <expression> is a <ground primary> with an <operator identifier> and the <result sort> of the <operator signature> is a <syntype> then the range check for that syntype defined in § 5.4.1.9.1 is applied to the result value. The value of the range check must be True.

If an <expression> contains an <extended primary> (that is a <synonym>, <indexed primary>, <field primary> or <structure primary>), this is replaced at the concrete syntax level as defined in § 5.4.2.3, § 5.4.2.4, § 5.4.2.5 and § 5.4.2.6 respectively before relationship to the abstract syntax is considered.

The optional <qualifier> before a <quoted operator> has the same relationship with the abstract syntax as a <qualifier> of an <operator identifier> (see § 5.2.2).

Semantics

A ground expression is interpreted as the value denoted by the ground term syntactically equivalent to the ground expression.

In general there is no need or reason to distinguish between the ground term and the value of the ground term. For example the ground term for the unity integer value can be written "1". Usually there are several ground terms which denote the same value, for instance the integer ground terms "0+1", "3-2" and "(7+5)/12", and it is usual to consider a simple form of ground term (in this case "1") as denoting the value.

A ground expression has a sort which is the sort of the equivalent ground term.

A ground expression has a value which is the value of the equivalent ground term.

5.4.2.3 *Synonym*

Concrete textual grammar

<synonym> ::=
<synonym identifier>
| <external synonym>

The alternative <external synonym> is described in § 4.3.1.

Semantics

A synonym is a shorthand for denoting an expression defined elsewhere.

Model

A <synonym> represents the <ground expression> defined by the <synonym definition> identified by the <synonym identifier>. An <identifier> used in the <ground expression> represents an *identifier* in the abstract syntax according to the context of the <synonym definition>.

5.4.2.4 *Indexed primary*

An indexed primary is a shorthand syntactic notation which can be used to denote "indexing" of an "array" value. However, apart from the special syntactic form an indexed primary has no special properties and denotes an operator with the primary as a parameter.

Concrete textual grammar

<indexed primary> ::=
<primary> (<expression list>)

Semantics

An indexed expression represents the application of an Extract! operator.

Model

A <primary> followed by a bracketed <expression list> is derived concrete syntax for the concrete syntax

Extract!(<primary>, <expression list>)

and then this is considered as a legal expression even though Extract! is not allowed as an operator name in the concrete syntax for expressions. The abstract syntax is determined from this concrete expression according to § 5.4.2.2.

5.4.2.5 *Field primary*

An field primary is a shorthand syntactic notation which can be used to denote "field selection" of "structures". However, apart from the special syntactic form an field primary has no special properties and denotes an operator with the primary as a parameter.

Concrete textual grammar

<field primary> ::=
<primary> <field selection>

<field selection> ::=
        ! <field name>
    | (<field name> { , <field name> }*)

The field name must be a field name defined for the sort of the primary.

Semantics

A field primary represents the application of one of the field extract operators of a structured sort.

Model

The form
        <primary> (<field name>)
is derived syntax for
        <primary> ! <field name>

The form
        <primary> (<first field name> { , <field name> }*)
is derived syntax for
        <primary> ! < first field name > { ! <field name> }*
where the order of field names is preserved.

The form
        <primary> ! <field name>
is derived syntax for
        <field extract operator name> (<primary>)

where the field extract operator name is formed from the concatenation of the field name and "Extract!" in that order. For example
        s ! f1
is derived syntax for
        f1Extract!(s)

and then this is considered as a legal expression even though f1Extract! is not allowed as an operator name in the concrete syntax for expressions. The abstract syntax is determined from this concrete expression according to § 5.4.2.2.

In the case where there is an operator defined for a sort so that
        Extract!(s,name)

is a valid term when "name" is the same as a valid field name of the sort of s then a primary
        s(name)

is derived concrete syntax for
        Extract!(s,name)
and the field selection must be written
        s ! name

5.4.2.6 *Structure primary*

Concrete textual grammar

<structure primary> ::=
    [<qualifier>] (. <expression list> .)

Semantics

A structure primary represents a value of a structured sort which is constructed from expressions for each field of the structure.

The form

(. <expression list> .)

is derived concrete syntax for

Make!(<expression list>)

where this is considered as a legal ground expression even though Make! is not allowed as an operator name in concrete syntax for ground expressions. The abstract syntax is determined from this concrete ground expression according to § 5.4.2.2.

5.4.2.7 *Conditional ground expression*

Concrete textual grammar

<conditional ground expression> ::=
IF <Boolean ground expression>
THEN <consequence ground expression>
ELSE <alternative ground expression>
FI

<consequence ground expression> ::=
<ground expression>

<alternative ground expression> ::=
<ground expression>

The <conditional ground expression> represents a *ground expression* in the abstract syntax. If the <Boolean ground expression> represents True then the *ground expression* is represented by the <consequence ground expression> otherwise it is represented by the <alternative ground expression>.

The sort of the <consequence ground expression> must be the same as the sort of the <alternative ground expression>.

Semantics

A conditional ground expression is a ground primary which is interpreted as either the consequence ground expression or the alternative ground expression.

If the <Boolean ground expression> has the value True then the <alternative ground expression> is not interpreted. If the <Boolean ground expression> has the value False then the <consequence ground expression> is not interpreted. The further behaviour of the system is undefined if the <ground expression> which is interpreted has the value of an error.

A conditional ground expression has a sort which is the sort of the consequence ground expression (and also the sort of the alternative ground expression).

5.5 *Use of data with variables*

This section defines the use of data and variables declared in processes and procedures, and the imperative operators which obtain values from the underlying system.

A variable has a sort and an associated value of that sort. The value associated with a variable may be changed by assigning a new value to the variable. The value associated with the variable may be used in an expression by accessing the variable.

Any expression containing a variable is considered to be "active" since the value obtained by interpreting the expression may vary according to the value last assigned to the variable.

5.5.1 *Variable and data definitions*

Concrete textual grammar

<data definition> ::=
{ <partial type definition>
| <syntype definition>
| <generator definition>
| <synonym definition> } <end>

A data definition forms part of a *data type definition* if it is a <partial type definition> or <syntype definition> as defined in § 5.2.1 and § 5.4.1.9 respectively. The rules <generator definition> and <synonym definition> are defined in § 5.4.1.12 and §5.4.1.13 respectively.

The syntax for introducing process variables and for procedure parameter variables is given in § 2.5.1.1 and § 2.3.4 respectively. A variable defined in a procedure must not be revealed.

Semantics

A data definition is used either for the definition of part of a data type or the definition of a synonym for an expression as further defined in § 5.2.1, § 5.4.1.9 or § 5.4.1.13.

When a variable is created it contains a special value called undefined which is distinct from any other value of the sort of that variable.

5.5.2 *Accessing variables*

The following defines how an expression involving variables is interpreted.

5.5.2.1 *Active expressions*

Abstract grammar

Active-expression = *Variable-access* |
Conditional-expression |
Operator-application |
Imperative-operator

Concrete textual grammar

<active expression> ::=
<active expression>

<active primary> ::=
<variable access>
| <operator application>
| <conditional expression>
| <imperative operator>
| (<active expression>)
| <active extended primary>

<active extended primary> ::=
<active extended primary>

<expression list> ::=
< expression> { , < expression> }*

To be concise the concrete syntax for <active expression> is given as <expression> in § 5.4.2.2.. An <expression> is an <active expression> if it contains an <active primary>.

Also to be concise the concrete syntax for <active extended primary> is given as <extended primary> in § 5.4.2.2. An <extended primary> is an <active extended primary> if it contains an <active primary>. For an <extended primary> replacement at the concrete syntax level takes place as defined in § 5.4.2.3, § 5.4.2.4, § 5.4.2.5 and § 5.4.2.6 before the relationship to the abstract syntax is considered.

Semantics

An active expression is an expression whose value will depend on the current state of the system.

An active expression has a sort which is the sort of the equivalent ground term.

An active expression has a value which is the ground term equivalent to the active expression at the time of interpretation.

Model

Each time the active expression is interpreted the value of the active expression is determined by finding the ground term equivalent to the active expression. This ground term is determined from a ground expression formed by replacing each active primary in the active expression by the ground term equivalent to the value of that active primary. The value of an active expression is the same as the value of the ground expression .

Within an active expression each operator is interpreted in the order determined either by the concrete syntax given in § 5.4.2.2 or in the case of ambiguity from left to right. Within an active expression list or expression list each element of the list is interpreted in the order left to right.

5.5.2.2 *Variable access*

Abstract grammar

Variable-access = *Variable-identifier*

Concrete textual grammar

<variable access> ::=
<variable identifier>

Semantics

A variable access is interpreted as giving the value associated with the identified variable.

A variable access has a sort which is the sort of the variable identified by the variable access.

A variable access has a value which is the value last associated with the variable or if that value was the special value "undefined" then an error. If the value of a variable access is an error then the further behaviour of the system is undefined.

5.5.2.3 *Conditional expression*

A conditional expression is an expression which is interpreted as either the consequence or the alternative.

Abstract grammar

Conditional-expression	::	*Boolean-expression* *Consequence-expression* *Alternative-expression*
Boolean-expression	=	*Expression*
Consequence-expression	=	*Expression*
Alternative-expression	=	*Expression*

The sort of the *consequence expression* must be the same as the sort of the *alternative expression.*

Concrete textual grammar

<conditional expression> ::=
IF <Boolean active expression>
THEN <consequence expression>
ELSE <alternative expression>
FI
| IF <Boolean expression>
THEN <active consequence expression>
ELSE <alternative expression>
FI
| IF <Boolean expression>
THEN <consequence expression>
ELSE <active alternative expression>
FI

<consequence expression> ::=
<expression>

<alternative expression> ::=
<expression>

A <conditional expression> is distinguished from a <conditional ground expression> by the occurrence of an <active expression> in the <conditional expression>.

Semantics

A conditional expression is interpreted as the interpretation of the condition followed by either the interpretation of the consequence expression or the interpretation of the alternative expression. The consequence is interpreted only if the condition has the value True, so that if the condition has the value False then the further behaviour of the system is undefined only if the alternative expression is an error. Similarly, the alternative is interpreted only if the condition has the value False, so that if the condition has the value True then the further behaviour of the system is undefined only if the consequence expression is an error.

The conditional expression has a sort which is the same as the sort of the consequence and alternative.
The conditional expression has a value which is the value of the consequence if the condition is True or the value of the alternative if the condition is False.

5.5.2.4 *Operator application*

An operator application is the application of an operator where one or more of the actual arguments is an active expression.

Abstract grammar

Operator-application	::	*Operator-identifier* *Expression*$^{+}$

If an argument *sort* of the *operator signature* is a *syntype* and the corresponding *expression* in the list of *expressions* is a *ground expression*, the range check defined in § 5.4.1.9.1 applied to the value of the *expression* must be True.

Concrete textual grammar

<operator application> ::=
<operator identifier> (<active expression list>)

<active expression list> ::=
<active expression> [, <expression list>]
| <ground expression> , <active expression list>

An <operator application> is distinguished from the syntactically similar <ground expression> by one of the <expression>s in the bracketed list of <expression>s being an <active expression>. If all the bracketed <expression>s are <ground expression>s then the construction represents a *ground expression* as defined in § 5.4.2.2.

Semantics

An operator application is a active expression which has the value of the ground term equivalent to the operator application. The equivalent ground term is determined as in § 5.5.2.1.

The list of expressions for the operator application are interpreted in the order given before interpretation of the operator.

If an argument sort of the operator signature is a syntype and the corresponding expression in the active expression list is an active expression then the range check defined in § 5.4.1.9.1 is applied to the value of the expression. If the range check is False at the time of interpretation then the system is in error and the further behaviour of the system is undefined.

If the result sort of the operator signature is a syntype then the range check defined in § 5.4.1.9.1 is applied to the value of the operator application. If the range check is False at the time of interpretation then the system is in error and the further behaviour of the system is undefined.

5.5.3 *Assignment statement*

Abstract grammar

Assignment-statement	::	*Variable-identifier* *Expression*

The sort of the *variable identifier* and the sort of the *expression* must be the same.

If the *variable* is declared with a *syntype* and the *expression* is a *ground expression*, then the range check defined in § 5.4.1.9.1 applied to the *expression* must be True.

Concrete textual grammar

<assignment statement> ::=
<variable> := <expression>

<variable> ::=
<variable identifier>
| <indexed variable>
| <field variable>

If the <variable > is a <variable identifier> then the <expression> in the concrete syntax represents the <expression> in the abstract syntax. The other forms of <variable>, <indexed variable> and <field variable>, are derived syntax and the <expression> in the abstract syntax is found from the equivalent concrete syntax defined in § 5.5.3.1 and § 5.5.3.2 below.

Semantics

An assignment statement is interpreted as creating an association from the variable identified in the assignment statement to the value of the expression in the assignment statement. The previous association of the variable is lost.

If the variable is declared with a syntype and the expression is an active expression, then the range check defined in § 5.4.1.9.1 is applied to the expression. If this range check is equivalent to False then the assignment is in error and the further behaviour of the system is undefined.

5.5.3.1 *Indexed variable*

An indexed variable is a shorthand syntactic notation which can be used to denote "indexing" of "arrays". However, apart from the special syntactic form an indexed active primary has no special properties and denotes an operator with the active primary as a parameter.

Concrete textual grammar

<indexed variable> ::=
<variable> (<expression list>)

There must be an appropriate definition of an operator named Modify!.

Semantics

An indexed variable represents the assignment of a value formed by the application of the Modify! operator to an access of the variable and the expression given in the indexed variable.

Model

The concrete syntax form
<variable> (<expression list>) := <expression>
is derived concrete syntax for
<variable> := Modify!(<variable>,<expression list>, <expression>)
where the same <variable> is repeated and the text is considered as a legal assignment even though Modify! is not allowed as an operator name in the concrete syntax for expressions. The abstract syntax is determined for this <assignment statement> according to § 5.5.3 above.

The model for indexed variables must be applied before the model for import (see § 4.13).

5.5.3.2 *Field variable*

A field variable is a shorthand for assigning a value to a variable so that only the value in one field of that variable has changed.

Concrete textual grammar

<field variable> ::=
<variable> <field selection>

There must be an appropriate definition of an operator named Modify!. Normally this definition will be implied by a structured sort definition.

Semantics

A field variable represents the the assignment of a value formed by the application of a field modify operator.

Model

Bracketed field selection is derived syntax for ! <field name> field selection as defined in § 5.4.2.5.

The concrete syntax form
<variable> ! <field name> := <expression>
is derived concrete syntax for
<variable> := <field modify operator name> (<variable>, <expression>)
where

a) the same <variable> is repeated, and

b) the <field modify operator name> is formed from the concatenation of the field name and "Modify!", and then

c) the text is considered as a legal assignment even though the <field modify operator name> is not allowed as an operator name in the concrete syntax for expressions.

If there is more than one <field name> in the field selection then they are modelled as above by expanding each ! <field name> in turn from right to left and considering the remaining part of the <field variable> as a <variable>. For example

var ! fielda ! fieldb := expression;

is first modelled by

var ! fielda := fieldbModify!(var ! fielda, expression);

and then by

var := fieldaModify!(var, fieldbModify!(var ! fielda, expression));

The abstract syntax is determined for the <assignment statement> formed by the modelling according to § 5.5.3 above.

5.5.3.3 *Default assignment*

A default assignment is shorthand for assigning the same value to all variables of a specified sort immediately after they are created.

Concrete textual grammar

<default assignment> ::=

DEFAULT <ground expression> [<end>]

A <partial type definition> or <syntype definition> must contain not more than one <default assignment>. (This prevents multiple assignments arising from generator instantiations).

Semantics

A default assignment is optionally added to a properties expression of a sort. A default assignment specifies that any variable declared with the sort introduced by the partial type definition or syntype definition is immediately assigned the value of the ground expression.

If there is no default assignment then when a variable is declared it will be associated with the undefined value.

A variable may be assigned an alternative value when it is declared by including an explicit assignment with the declaration.

Default assignments are not inherited.

Model

The concrete syntax form

DEFAULT <ground expression>

used in a properties expression where the sort s is introduced implies an assignment of the <ground expression> to a variable. This assignment is interpreted immediately after the declaration of the variable and before any explicitly specified action in the same process or procedure is interpreted. For example if

DEFAULT 2*dnumber

is given for sort s and there is a declaration in the concrete syntax

DCL v s;

then there is an implied assignment

v := 2*dnumber;

If the declaration also has an <initial value> then the <initial value> is assigned to the variable after the <ground expression> in the <default assignment>.

The implied assignment statement has the normal relationship of an <assignment statement> to the abstract syntax (see § 5.5.3).

If a <default assignment> is specified for a <data definition> then the <sort> (representing a syntype or sort) has a default assignment value which is the value of the <ground expression> of the <default assignment>. If no <default assignment> is given in <syntype definition> then the syntype has a default assignment value if the *parent sort identifier* (identifying a syntype or sort) given in the *syntype definition* has a default assignment value.

For a <syntype definition> the assignments are interpreted if and only if the range check as defined in § 5.4.1.9.1 gives True when applied to the default assignment value. That is, for each variable of the syntype there is an implied decision of the form

DECISION <range check>;
(True) : <default assignment>
ELSE: ENDDECISION.

5.5.4 *Imperative operators*

Imperative operators obtain values from the underlying system state.

Abstract grammar

Imperative-operator	=	*Now-expression* |
		Pid-expression |
		View-expression |
		Timer-active-expression

Concrete textual grammar

<imperative operator> ::=
<now expression>
| <import expression>
| <PId expression>
| <view expression>
| <timer active expression>

The alternative <import expression> is defined in § 4.13.

Imperative operators are expressions for checking whether timers are active or for accessing the system clock, the PId values associated with a process or imported variables.

5.5.4.1 *NOW*

Abstract grammar

Now-expression	::	()

Concrete textual grammar

<now expression> ::=
NOW

Semantics

The now expression is an expression which accesses a system clock variable to determine the absolute system time.

The now expression represents an expression requesting the current value of the system clock giving the time. The origin and unit of time are system dependent. Whether two occurences of NOW in the same transition will give the same value is system dependent.

A now expression has the time sort.

5.5.4.2 *IMPORT expression*

Concrete textual grammar

The concrete syntax for an import expression is defined in § 4.3.

Semantics

In addition to the semantics defined in § 4.13 an import expression is interpreted as a variable access (see § 5.5.2.2) to the implicit variable for the import expression.

Model

The import expression has implied syntax for the importing of the value as defined in § 4.13 and also has an implied *variable access* of the implied variable for the import in the context where the <import expression> appears.

5.5.4.3 *PId expression*

Abstract grammar

Pid-expression	=	*Self-expression* | *Parent-expression* | *Offspring-expression* | *Sender-expression*
Self-expression	::	()
Parent-expression	::	()
Offspring-expression	::	()
Sender-expression	::	()

Concrete textual grammar

<PId expression> ::=
SELF
| PARENT
| OFFSPRING
| SENDER

Semantics

A PId expression accesses one of the implicit process variables defined in § 2.4.4. The process variable expression is interpreted as the last value associated with the corresponding implicit variable.

A PId expression has a sort which is PId.

A PId expression has a value which is the last value associated with the corresponding variable as defined by § 2.4.4.

5.5.4.4 *View expression*

A view expression allows a process to obtain the value of a variable of another process in the same block as if the variable were defined locally. The viewing process can not modify the value associated with the variable.

Abstract grammar

View-expression	::	*Variable-identifier*
		Expression

The *expression* must be a PId expression.

The *variable identifier* must be one of the identifiers of one of the variables in the process identified by the *expression.*

Concrete textual grammar

<view expression> ::=
VIEW (<variable identifier>, <PId expression>)

The <variable identifier> must be defined to be viewed in a <view definition> in the process containing the <view expression>. The <qualifier> in <variable identifier> may be omitted only if no other variables with the same <name> part are contained in a <view definition> for the enclosing <process definition>.

Semantics

A view expression is interpreted in the same way as a variable access (see § 5.5.2.2). The variable accessed is the variable in the process identified by the PId expression which corresponds to the PId expression (see § 5.5.4.3).

A view expression has a value and a sort which are the value and sort of the variable access.

The PId expression must identify an existing process in the same block as the process in which the view expression is interpreted otherwise the view expression is in error and the further behaviour of the system is undefined. The PId expression must identify the same process type as *Process-identifier* in the corresponding view definition.

5.5.4.5 *Timer active expression*

Abstract grammar

Timer-active-expression :: *Timer-identifier*
*Expression**

The sorts of the *Expression** in the *Timer-active-expression* must correspond by position to the *Sort-reference-identifier** directly following the *Timer-name* (§ 2.8) identified by the *Timer-identifier*.

Concrete textual grammar

<Timer active expression> ::=
ACTIVE (<timer identifier> [(<expression list>)])

<expression list> is defined in § 5.5.2.1.

Semantics

A timer active expression is an expression of the Boolean sort which has the value True if the timer identified by timer identifier, and set with the same values as denoted by the expression list (if any), is active (see §2.8.2). Otherwise the timer active expression has the value False. The expressions are interpreted in the order given.

5.6 *Predefined data*

This section defines data sorts and data generators implicitly defined at system level. Note that section 5.4.1.1 defines the syntax and precedence of special operators (infix and monadic), but the semantics of these operators (except REM and MOD) are defined by the data definitions in this section.

5.6.1 *Boolean sort*

5.6.1.1 *Definition*

```
NEWTYPE     Boolean
        LITERALS    True,False;
        OPERATORS

                "NOT"    : Boolean              -> Boolean;
                                                        /*
                 "="     : Boolean, Boolean     -> Boolean;     The "=" and "/=" operators
                "/="     : Boolean, Boolean     -> Boolean;     are implied. See § 5.4.1.4
                                                        */
                "AND"    : Boolean, Boolean     -> Boolean;
                "OR"     : Boolean, Boolean     -> Boolean;
                "XOR"    : Boolean, Boolean     -> Boolean;
                "=>"     : Boolean, Boolean     -> Boolean;
        AXIOMS
                "NOT"(True )          == False;
                "NOT"(False)          == True ;

                "="  ( True, True)    == True ;
                "="  ( True,False)    == False;
                "="  (False, True)    == False;
                "="  (False,False)    == True ;

                "/=" ( True, True)    == False;
                "/=" ( True,False)    == True ;
                "/=" (False, True)    == True ;
                "/=" (False,False)    == False;

                "AND"( True, True) == True ;
                "AND"( True,False) == False;
                "AND"(False, True) == False;
                "AND"(False,False) == False;

                "OR" ( True, True)    == True ;
                "OR" ( True,False)    == True ;
                "OR" (False, True)    == True ;
                "OR" (False,False)    == False;

                "XOR"( True, True) == False;
                "XOR"( True,False) == True ;
                "XOR"(False, True) == True ;
                "XOR"(False,False) == False;

                "=>" ( True, True)    == True ;
                "=>" ( True,False)    == False;
```

```
        "=>" (False, True)   == True ;
        "=>" (False,False)   == True ;
ENDNEWTYPE Boolean;
```

5.6.1.2 *Usage*

The Boolean sort is used to represent true and false values. Often it is used as the result of a comparison.

The Boolean sort is used by many of the short-hand forms of data in SDL such as axioms without the "==" symbol, and the implicit equality operators "=" and "/=".

5.6.2 *Character sort*

5.6.2.1 *Definition*

```
NEWTYPE Character

    LITERALS
        NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL,
        BS,  HT,  LF,  VT,  FF,  CR,  SO,  SI,
        DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB,
        CAN, EM,  SUB, ESC, IS4, IS3, IS2, IS1,
        ' ',  '!',  '"',  '#',  '¤',  '%',  '&',  '''',
        '(',  ')',  '*',  '+',  ',',  '-',  '.',  '/',
        '0',  '1',  '2',  '3',  '4',  '5',  '6',  '7',
        '8',  '9',  ':',  ';',  '<',  '=',  '>',  '?',
        '@',  'A',  'B',  'C',  'D',  'E',  'F',  'G',
        'H',  'I',  'J',  'K',  'L',  'M',  'N',  'O',
        'P',  'Q',  'R',  'S',  'T',  'U',  'V',  'W',
        'X',  'Y',  'Z',  '[',  '\',  ']',  '^',  '_',
        '`',  'a',  'b',  'c',  'd',  'e',  'f',  'g',
        'h',  'i',  'j',  'k',  'l',  'm',  'n',  'o',
        'p',  'q',  'r',  's',  't',  'u',  'v',  'w',
        'x',  'y',  'z',  '{',  '|',  '}',  '~',  DEL;
        /*  '"' is an apostrophe, ' ' is a space, '~' is an overline or tilde */
    OPERATORS
                                                    /*
        "="  : Character, Character  -> Boolean;        The "=" and "/=" operator signatures
        "/=" : Character, Character  -> Boolean;        are implied – see § 5.4.1.4
                                                    */
        "<"  : Character, Character  -> Boolean;
        "<=": Character, Character  -> Boolean;
        ">"  : Character, Character  -> Boolean;
        ">=": Character, Character  -> Boolean;
    AXIOMS
    /* the following specifies "less than" between adjacent character literals*/
        NUL < SOH == True;      SOH < STX == True;
        STX < ETX == True;      ETX < EOT == True;
        EOT < ENQ == True;      ENQ < ACK == True;
        ACK < BEL == True;      BEL < BS  == True;
        BS  < HT  == True;      HT  < LF  == True;
        LF  < VT  == True;      VT  < FF  == True;
        FF  < CR  == True;      CR  < SO  == True;
        SO  < SI  == True;      SI  < DLE == True;
```

```
DLE < DC1 == True;      DC1 < DC2 == True;
DC2 < DC3 == True;      DC3 < DC4 == True;
DC4 < NAK == True;      NAK < SYN == True;
SYN < ETB == True;      ETB < CAN == True;
CAN < EM  == True;      EM  < SUB == True;
SUB < ESC == True;      ESC < IS4 == True;
IS4 < IS3 == True;      IS3 < IS2 == True;
IS2 < IS1 == True;      IS1 < ' ' == True;
' ' < '!' == True;      '!' < '"' == True;
'"' < '#' == True;      '#' < '¤' == True;
'¤' < '%' == True;      '%' < '&' == True;
'&' < ''' == True;      ''' < '(' == True;
'(' < ')' == True;      ')' < '*' == True;
'*' < '+' == True;      '+' < ',' == True;
',' < '-' == True;      '-' < '.' == True;
'.' < '/' == True;      '/' < '0' == True;
'0' < '1' == True;      '1' < '2' == True;
'2' < '3' == True;      '3' < '4' == True;
'4' < '5' == True;      '5' < '6' == True;
'6' < '7' == True;      '7' < '8' == True;
'8' < '9' == True;      '9' < ':' == True;
':' < ';' == True;      ';' < '<' == True;
'<' < '=' == True;      '=' < '>' == True;
'>' < '?' == True;      '?' < '@' == True;
'@' < 'A' == True;      'A' < 'B' == True;
'B' < 'C' == True;      'C' < 'D' == True;
'D' < 'E' == True;      'E' < 'F' == True;
'F' < 'G' == True;      'G' < 'H' == True;
'H' < 'I' == True;      'I' < 'J' == True;
'J' < 'K' == True;      'K' < 'L' == True;
'L' < 'M' == True;      'M' < 'N' == True;
'N' < 'O' == True;      'O' < 'P' == True;
'P' < 'Q' == True;      'Q' < 'R' == True;
'R' < 'S' == True;      'S' < 'T' == True;
'T' < 'U' == True;      'U' < 'V' == True;
'V' < 'W' == True;      'W' < 'X' == True;
'X' < 'Y' == True;      'Y' < 'Z' == True;
'Z' < '[' == True;      '[' < '\' == True;
'\' < ']' == True;      ']' < '^' == True;
'^' < '_' == True;      '_' < '`' == True;
'`' < 'a' == True;      'a' < 'b' == True;
'b' < 'c' == True;      'c' < 'd' == True;
'd' < 'e' == True;      'e' < 'f' == True;
'f' < 'g' == True;      'g' < 'h' == True;
'h' < 'i' == True;      'i' < 'j' == True;
'j' < 'k' == True;      'k' < 'l' == True;
'l' < 'm' == True;      'm' < 'n' == True;
'n' < 'o' == True;      'o' < 'p' == True;
'p' < 'q' == True;      'q' < 'r' == True;
'r' < 's' == True;      's' < 't' == True;
't' < 'u' == True;      'u' < 'v' == True;
'v' < 'w' == True;      'w' < 'x' == True;
'x' < 'y' == True;      'y' < 'z' == True;
'z' < '{' == True;      '{' < '|' == True;
'|' < '}' == True;      '}' < '~' == True;
```

```
'-'    < DEL  == True;
FOR ALL a,b,c  IN Character (
      a < a                        == False;
      a < b AND b < c => a < c     == True;
      a < b                        == b > a;
      a < b OR  a > b              == a /= b;
      a < b = > NOT (b < a);
      NOT ( a /= b )               == a  = b;
      a < b OR  a = b              == a <= b;
      a > b OR  a = b              == a >= b;)
```

ENDNEWTYPE Character;

5.6.2.2 *Usage*

The Character sort defines character strings of length 1, where the characters are those of the International Alphabet No. 5. These are defined either as strings or as abbreviations according the International Reference Version of the alphabet. The printed representation may vary according to national usage of the alphabet.

There are 128 different literals and values defined for Character. The ordering of the values and equality and inequality are defined.

5.6.3 *String generator*

5.6.3.1 *Definition*

```
GENERATOR String(TYPE Itemsort, LITERAL Emptystring) /*Strings are "indexed" from one */
      LITERALS  Emptystring;
      OPERATORS
          MkString : Itemsort -> String;                  /* make a string from an item */
          Length   : String   -> Integer;                 /* length of string */
          First    : String   -> Itemsort;                /* first item in string*/
          Last     : String   -> Itemsort;                /* last item in string */
          "//"     : String, String      -> String;       /* concatenation */
          Extract! : String, Integer     -> Itemsort;     /* get item from string */
          Modify!  : String, Integer, Itemsort -> String;/* modify value of string */
          SubString: String, Integer,Integer   -> String;/* get substring from string */
              /*substring (s,i,j) gives a string of length j starting from the ith element */
      AXIOMS
      FOR ALL item, itemi, itemj, item1, item2  IN Itemsort (
      FOR ALL s, s1, s2, s3 IN String (
      FOR ALL i, j IN Integer (
          type String Length(Emptystring)          == 0;
          type String Length(MkString(item))       == 1;
          type String Extract!(MkString(item),1)   == item;
          First(s)                                 == Extract!(s,1);
          Last (s)                                 == Extract!(s,Length(s));
          Length( s1 // s2 )                       == Length (s1) + Length (s2) ;
          Length(Modify!(s,i,item))                == Length(s);
          ( s1 // s2  ) // s3                      == s1 // (s2 // s3);
          Emptystring // s                         == s;
          s // Emptystring                         == s;
          Emptystring = (MkString(item) // s2) == False;
          (MkString(item1) // s1) = (MkString (item2) // s2) == (item1 = item 2) AND (s1 = s2);
```

```
        i > 0 AND i <= Length(s) == True ==>
                        Extract!(Modify!(s,i,item),i)  == item;
        i /= j AND i > 0 AND i <= Length(s) AND j > 0 AND j <= Length(s) == True ==>
                        Extract!(Modify!(s,i,item),j)  == Extract!(s,j);
        i <= 0 OR i > Length(s) == True ==> Extract!(s,i) == ERROR!;

        i /= j == True ==>
            Modify!(Modify!(s,i,itemi),j,itemj) == Modify!(Modify!(s,j,itemj),i,itemi);
        Modify!(Modify!(s,i,item1),i,item2) == Modify!(s,i,item2);
        i <= 0 OR i > Length(s) == True ==> Modify!(s,i,item) == ERROR!;

        i <=  Length(s1) == True  ==>
                        Extract!(s1 // s2, i)   == Extract!(s1,i);
        i  >  Length(s1) == True  ==>
                        Extract!(s1 // s2, i)   == Extract!(s2,i - Length(s1));

        i > 0 AND i <= Length(s) == True ==> SubString(s,i,0) == Emptystring;
        i > 0 AND i <= Length(s) == True ==> SubString(s,i,1) == MkString(Extract!(s,i));
        i > 0 AND i <= Length(s) AND i -1+j <= Length(s) AND j > 1 == True ==>
                        SubString(s,i,j) == SubString(s,i,1) // SubString(s,i+1,j-1);
        i < 0 OR i > Length(s) OR j <= 0 OR i+j > Length(s) == True ==>
                        SubString(s,i,j) == ERROR!;

        i > 0 AND i <= Length(s)  == True ==>
                    Modify!(s,i,item)    ==
                        Substring(s,1,i-1) // MkString(item) // Substring(s,i+1,Length(s)-i);)));
ENDGENERATOR String;
```

5.6.3.2 *Usage*

A string generator can be used to define a sort which allows strings of any item sort to be constructed. The most common use will be for the Charstring defined below.

The Extract! and Modify! operators will typically be used with the shorthands defined in § 5.4.2.4 and § 5.5.3.1 for accessing the values of strings and assigning values to strings.

5.6.4 *Charstring sort*

5.6.4.1 *Definition*

```
NEWTYPE Charstring String (Character, '')
        ADDING LITERALS NAMECLASS "" (  ( ' ':'&')  OR  """"  OR  ( '(': '¯' )  )+ "";
        /* character strings of any length of any characters from a space ' ' to an overline '¯' */
        /*  equations of the form
            'ABC' == 'AB' // 'C';
            are implied – see § 5.4.1.2 */
        MAP     FOR ALL c IN Character LITERALS (
                        FOR ALL charstr IN Charstring LITERALS (
                            Spelling( charstr ) == Spelling( c ) ==> charstr == Mkstring(c);
                )       ); /* string 'A' is formed from character 'A' etc. */
ENDNEWTYPE Charstring;
```

5.6.4.2 *Usage*

The Charstring sort defines strings of characters. A Charstring literal can contain printing characters and spaces. A non printing character can be used as a string by using Mkstring, for example Mkstring(DEL).

/* Example */ SYNONYM newline_prompt Charstring = Mkstring(CR) // Mkstring(LF) // '$>';

5.6.5 *Integer sort*

5.6.5.1 *Definition*

```
NEWTYPE Integer
    LITERALS NAMECLASS ('0':'9')* ('0':'9') ;
        /* optional number sequence before one of the numbers 0 to 9 */
    OPERATORS
        "-"  : Integer              -> Integer;
        "+"  : Integer, Integer     -> Integer;
        "-"  : Integer, Integer     -> Integer;
        "*"  : Integer, Integer     -> Integer;
        "/"  : Integer, Integer     -> Integer;
                                       /*
        "="  : Integer, Integer     -> Boolean;     The "=" and "/=" operator signatures
        "/=" : Integer, Integer     -> Boolean;     are implied – see § 5.4.1.4
                                       */
        "<"  : Integer, Integer     -> Boolean;
        ">"  : Integer, Integer     -> Boolean;
        "<=": Integer, Integer      -> Boolean;
        ">=": Integer, Integer      -> Boolean;
        Float: Integer              -> Real;    /* axioms in NEWTYPE Real definition */
        Fix  : Real                 -> Integer; /* axioms in NEWTYPE Real definition */
    AXIOMS
    FOR ALL a, b, c IN Integer (
        /*negation*/
        0 - a                       ==  - a;
        /* addition*/
        0 + a                       ==  a;
        a + b                       ==  b + a;
        a + (b + c)                 ==  (a + b) + c;
        /*subtraction*/
        a - a                       ==  0;
        (a - b) - c                 ==  a - (b + c);
        (a - b) + c                 ==  (a + c) - b;
        a - (b - c)                 ==  (a + c) - b;
        /*multiplication*/
        a * 0                       ==  0;
        a * 1                       ==  a;
        a * b                       ==  b * a;
        a * (b * c)                 ==  (a * b) * c;
        a * (b + c)                 ==  a * b + a * c;
        a * (b - c)                 ==  a * b - a * c;
        /*ordering*/
        a + 1 > a                   ==  True;
        a  - 1 < a                  ==  True;
        /*equality*/
```

```
        (a>b) OR (b>a) == NOT (a=b);
        /* normal ordering axioms */
        "<"(a,a)                        ==  False;
        "<"(a,b)                        ==  ">"(b,a);
        "<="(a,b)                       ==  "OR"("<"(a,b),"="(a,b));
        ">="(a,b)                       ==  "OR"(">"(a,b),"="(a,b));
        "<"(a,b) == True                    ==>     NOT("<"(b,a)) == True;
        "<"(a,b) AND "<"(b,c) == True ==>       "<"(a,c)        == True ;
        /*division*/
        a / 0                           ==  ERROR!;
        a >= 0 AND b > a == True                    ==> a / b == 0;
        a >= 0 AND b <= a AND b > 0 == True  ==> a / b == 1 + (a-b)/b;
        a >= 0 AND b < 0 == True                    ==> a / b == - (a/(-b));
        a < 0  AND b < 0 == True                    ==> a / b == (-a)/(-b);
        a < 0  AND b > 0 == True                    ==> a / b == -( (-a)/b );
        /* Literals 2 to 9 */
        TYPE Integer 2 == 1 + 1; TYPE Integer 3 == 2 + 1;
        TYPE Integer 4 == 3 + 1; TYPE Integer 5 == 4 + 1;
        TYPE Integer 6 == 5 + 1; TYPE Integer 7 == 6 + 1;
        TYPE Integer 8 == 7 + 1; TYPE Integer 9 == 8 + 1;
    MAP /* Literals other than 0 to 9 */
        FOR ALL a,b,c IN Integer LITERALS
        ( Spelling(a) == Spelling(b) // Spelling(c), Length(Spelling(c)) == 1 ==>
                a  == b * ( 9 + 1) + c ;
        );
ENDNEWTYPE Integer;
```

5.6.5.2 *Usage*

The Integer sort is used for mathematical integers with decimal notation.

5.6.6 *Natural syntype*

5.6.6.1 *Definition*

```
SYNTYPE Natural = Integer CONSTANTS >= 0 ENDSYNTYPE Natural;
```

5.6.6.2 *Usage*

The natural syntype is used when positive integers only are required. All operators will be the integer operators but when a value is used as a parameter or assigned the value is checked. A negative value will be an error.

5.6.7 *Real sort*

5.6.7.1 *Definition*

```
NEWTYPE Real
    LITERALS NAMECLASS ( ('0':'9')* ('0':'9') ) OR ( ('0':'9')* '.' ('0':'9')+ );
    OPERATORS
        "-"  : Real             -> Real;
        "+"  : Real, Real       -> Real;
        "-"  : Real, Real       -> Real;
        "*"  : Real, Real       -> Real;
        "/"  : Real, Real       -> Real;
```

```
                                        /*
    "="  : Real, Real          -> Boolean;      The "=" and "/=" operator signatures
    "/=" : Real, Real          -> Boolean;      are implied – see § 5.4.1.4
                                        */
    "<"  : Real, Real          -> Boolean;
    ">"  : Real, Real          -> Boolean;
    "<=": Real, Real           -> Boolean;
    ">=": Real, Real           -> Boolean;
AXIOMS
FOR ALL a, b, c IN Real (
    /*negation*/
    0 - a                      ==  - a;
    /* addition*/
    0 + a                      ==  a;
    a + b                      ==  b + a;
    a + (b + c)                ==  (a + b) + c;
    /*subtraction*/
    a - a                      ==  0;
    (a - b) - c                ==  a - (b + c);
    (a - b) + c                ==  (a + c) - b;
    a - (b - c)                ==  (a + c) – b;
    /*multiplication*/
    a * 0                      ==  0;
    a * 1                      ==  a;
    a * b                      ==  b * a;
    a * (b * c)                ==  (a * b) * c;
    a * (b + c)                ==  a * b + a * c;
    a * (b - c)                ==  a * b - a * c; );
    /*ordering*/
    FOR ALL i, j IN Integer (
       Float(i) > Float(j)     ==  TYPE Integer ">"(i,j);
       Float(j) = 0 == False   ==> Float(i) / Float(j) > 0 == Float(i) > 0 AND Float(j) > 0
                                                  OR Float(i) < 0 AND Float(j) < 0;
       Float(i) > 0 AND Float(j) > 0 AND Float(i) > Float(j)
                               ==> Float(i) / Float(j) > 1 == True;);
       FOR ALL a, r, b IN Real (a + r < b + r == a < b;
          r > 0 ==> a * r < b * r == a < b;
          r < 0 ==> a * r < b * r == b < a;);
       /* normal ordering axioms */
       FOR ALL a, b, c, d IN Real (
         /* equality and ordering */
          (a>b) OR (b>a) == NOT (a=b);
          "<"(a,a)             ==  False;
          "<"(a,b)             ==  ">"(b,a);
          "<="(a,b)            ==  "OR"("<"(a,b),"="(a,b));
          ">="(a,b)            ==  "OR"(">"(a,b),"="(a,b));
          "<"(a,b) == True          ==>   NOT("<"(b,a)) == True;
          "<"(a,b) AND "<"(b,c) == True ==>  "<"(a,c)       == True ;
       /*division*/
          a / 0                ==  ERROR!;
          a = 0 == False ==> a / a  ==      1;
          a = 0 == False ==> 0 / a  ==      0;
          b = 0 == False ==> (a / b ) * b  == a;
```

```
        b = 0 OR c = 0 == False ==> (a * b) / (c * b) == a / c;
        b = 0 OR d = 0 == False ==> a / b + c / d == (a * d + b * c) / (b * d);
        b = 0 OR d = 0 == False ==> a / b  - c / d == (a * d  - b * c) / (b * d);
        b = 0 OR d = 0 == False ==>(a/b) * (c/d) == (a * c) / (b * d);
        b = 0 OR d = 0 == False ==> (a/b) / (c/d) == (a * d) / (b * c););
    /* conversions between integer and real */
    FOR ALL a, i, j IN Integer (
    FOR ALL  r IN Real  (
      Fix(Float(a))            ==   a;
      r - 1.0 < Float(Fix(r))  ==   True; /* Note Fix(1.5) == 1, Fix(-0.5) == -1 */
      Float(Fix(r)) <= r       ==   True;
      Float(TYPE integer "+"(i,j)) ==        Float(i) + Float(j);));
MAP
    FOR ALL r,s IN Real LITERALS (
        FOR ALL i,j IN Integer LITERALS (
            Spelling(r) == Spelling(i) ==> r == Float(i);
            Spelling(r) == Spelling(i) ==> i == Fix(r);
            Spelling(r) == Spelling(i) // Spelling(s), Spelling(s) == '.' // Spelling(j)
                                       ==> r == Float(i) + s;
            Spelling(r) == '.' // Spelling(i), Length(Spelling(i)) == 1
                                       ==> r == Float(i) / 10;
            Spelling(r) == '.' // Spelling(i) // Spelling(j), Length(Spelling(i)) == 1,
                Spelling(s) == '.' // Spelling(j)
                                       ==> r == (Float(i) + s) / 10;
    )   );
ENDNEWTYPE Real;
```

5.6.7.2 *Usage*

The real sort is used to represent real numbers. The real sort can represent all numbers which can be represented as one integer divided by another. Numbers which cannot be represented in this way (irrational numbers – for example $\sqrt{2}$) are not part of the real sort. However for practical engineering a sufficiently accurate approximation can usually be used. Defining a set of numbers which includes all irrationals is not possible without using additional techniques.

5.6.8 *Array generator*

5.6.8.1 *Definition*

```
GENERATOR Array (TYPE Index, TYPE Itemsort)
    OPERATORS
        Make!    : Itemsort                -> Array ;
        Modify!  : Array,Index,Itemsort    -> Array ;
        Extract! : Array,Index             -> Itemsort ;
    AXIOMS
    FOR ALL item, item1, item2, itemi, itemj  IN  Itemsort  (
    FOR ALL i, j, ipos  IN Index (
    FOR ALL a, s IN Array  (
        type Array Extract!(Make!(item,i)) == item ;
        Modify!(Modify!(s,i,item1),i,item2) == Modify!(s,i,item2);
        Extract!(Modify!(a,ipos,item),ipos)  == item ;
        i = j == False        ==> Extract!(Modify!(a,j,item),i) == Extract!(a,i);
        i = j == False        ==>
            Modify!(Modify!(s,i,itemi),j,itemj) == Modify!(Modify!(s,j,itemj),i,itemi);)));
      /*equality*/
```

```
        type Array Make! (item1) = Make! (item2)     == item1 = item2;
        Modify! (a,i,item) =s                        == (Extract! (s,i) = item) AND (a=s);
ENDGENERATOR Array;
```

5.6.8.2 *Usage*

The array generator can be used to define one sort which is indexed by another. For example

```
NEWTYPE indexbychar Array(Character,Integer)
ENDNEWTYPE indexbychar;
```

defines an array containing integers and indexed by characters.

Arrays are usually used in combination with the shorthand forms of Modify! and Extract! defined in § 5.5.3.1 and § 5.4.2.4 for indexing. For example

```
DCL charvalue indexbychar;
.........
TASK charvalue('A') := charvalue('B')-1;
```

5.6.9 *Powerset generator*

5.6.9.1 *Definition*

```
GENERATOR Powerset ( TYPE Itemsort )
    LITERALS Empty;
    OPERATORS
        "IN"    : Itemsort,    Powerset -> Boolean;   /* is member of operator          */
        Incl    : Itemsort,    Powerset -> Powerset;  /* include item in set            */
        Del     : Itemsort,    Powerset -> Powerset;  /* delete item from set           */
        "<"     : Powerset,    Powerset -> Boolean;   /* is proper subset of operator   */
        ">"     : Powerset,    Powerset -> Boolean;   /* is proper superset of operator */
        "<="    : Powerset,    Powerset -> Boolean;   /* is subset of operator          */
        ">="    : Powerset,    Powerset -> Boolean;   /* is superset of operator        */
        "AND"   : Powerset,    Powerset -> Powerset;  /* intersection of sets           */
        "OR"    : Powerset,    Powerset -> Powerset;  /* union of sets                  */
    AXIOMS
    FOR ALL  i, j IN Itemsort  (
    FOR ALL p, ps, a, b, c  IN Powerset  (
        i IN type Powerset Empty            == False;
        i IN Incl(i,ps)                     == True;
        i IN ps                             == i IN Incl(j,ps);
        type Powerset Del(i,Empty)          == Empty ;
        NOT(i IN ps)                        == Del(i,ps) = ps;
        Del(i,Incl(i,ps))                   == ps;
        i = j == False   ==> Del(i,Incl(j,ps))== Incl(j,Del(i,ps));
        Incl(i,Incl(j,p))                   == Incl(j,Incl(i,p));
        Incl(i,Incl(i,p))                   == Incl(i,p);
        a<b => (i IN a => i IN b)            == True;
        i IN ( a AND b )                    == TYPE Boolean "AND"( i IN a, i IN b);
        i IN ( a OR b )                     == TYPE Boolean "OR"( i IN a, i IN b);
        /*equality*/
        Empty = Incl (i,ps)  == False;
        Incl (i,a) = b       == (i IN b) AND (a=Del (i,b));
        /* normal ordering axioms */
```

```
        "<"(a,a)   ==   False;
        "<"(a,b)   ==   ">"(b,a);
        "<="(a,b) ==   "OR"("<"(a,b),"="(a,b));
        ">="(a,b) ==   "OR"(">"(a,b),"="(a,b));
        "<"(a,b) == True     ==> NOT("<"(b,a)) == True;
        TYPE Boolean "AND"("<"(a,b),"<"(b,c)) == True ==> "<"(a,c) == True ;))
ENDGENERATOR Powerset;
```

5.6.9.2 *Usage*

Powersets are used to represent mathematical sets. For example

```
    NEWTYPE Boolset Powerset(Boolean) ENDNEWTYPE Boolset;
```

can be used for a variable which can be empty or contain (True), (False) or (True,False).

5.6.10 *PId sort*

5.6.10.1 *Definition*

```
NEWTYPE PId
    LITERALS Null ;
    OPERATORS       unique! : PId -> PId ;

                    /*
                    "=" : Pid,Pid -> Boolean; The "=" and "/=" operator signatures
                    "/=": Pid,Pid -> Boolean; are implied – see § 5.4.1.4
                    */
    AXIOMS
    FOR ALL p, p1, p2 IN PId (
        unique! (p) = Null              == False;
        unique! (p1) = unique! (p2)     == p1 = p2  );
DEFAULT Null ;
ENDNEWTYPE PId;
```

5.6.10.2 *Usage*

The PId sort is used for process identities. Note that there are no other literals than the value Null. When a process is created the underlying system uses the unique! operator to generate a new unique value.

5.6.11 *Duration sort*

5.6.11.1 *Definition*

```
NEWTYPE Duration INHERITS Real ( "+", "-" ,">")
ADDING
    OPERATORS
        "*"   : Duration, Real -> Duration;
        "/"   : Duration, Real -> Duration;
    AXIOMS /* in the following every d must be a duration value from context */
    FOR ALL  d, z  IN Duration  (
    FOR ALL  r  IN  Real  (
        /*equality*/
        (d>z) OR (z>d) == NOT (d=z);
        /*  Duration multiplied by Real  */
        d * 0       == 0;
        0 * r       == 0;
        d * TYPE Real "+"(1,r)   == d + (d * r);
```

```
        d * TYPE Real "-" (1,r)   == d - (d * r);
        d * TYPE Real "-" (r,1)   == (d * r) - d;
        d * TYPE Real "-" (r)     == 0 - (d * r);
        /* Duration divided by Real */
        d / 0 == ERROR!;
        r = 0 == False ==> d / r == d * TYPE Real "/" (1,r) ;
        /* that is division is the same as multiplying by the (real) reciprocal */
        r = 0 == False ==> z * r = d    == ( d / r) = z;));
    MAP
        FOR ALL d IN Duration LITERALS (
        FOR ALL r IN Real LITERALS ( Spelling(d) == Spelling(r) ==> d == 1 * r ));
ENDNEWTYPE Duration;
```

5.6.11.2 *Usage*

The duration sort is used for the value to be added to the current time to set timers. The literals of the sort duration are the same as the literals for the real sort. The meaning of one unit of duration will depend on the system being defined.

Durations can be multiplied and divided by reals.

5.6.12 *Time sort*

5.6.12.1 *Definition*

```
NEWTYPE Time INHERITS Real OPERATORS ("<", "<=", ">", ">=") ADDING
    OPERATORS
        "+"  : Time, Duration     -> Time;
        "-"  : Time, Duration     -> Time;
        "-"  : Time, Time         -> Duration;
    AXIOMS
    FOR ALL t , t1, t2 IN Time (
    FOR ALL d,d1,d2 IN Duration (
        (t1>t2) OR (t2>t1) == NOT (t1=t2);
        t + 0                  == t;
        t - d                  == t + TYPE Duration "-"( 0 , d );
        (t + d1) + d2          == t + TYPE Duration "+"(d1,d2);
        (t + d1) - (t + d2 )   == TYPE Duration "-"(d1,d2);));
    MAP
        FOR ALL d IN Duration LITERALS (
        FOR ALL t IN Time LITERALS ( Spelling(d) == Spelling(t) ==> t == 0 + d ));
ENDNEWTYPE Time;
```

5.6.12.2 *Usage*

The NOW expression returns a value of the time sort. A time value may have a duration added or subtracted from it to give another time. A time value subtracted from another time value gives a duration. Time values are used to set the expiry time of timers.

The origin of time is system dependent. A unit of time is the amount of time represented by adding one duration unit to a time.

Appendix I
(To Recommendation Z.100)

The formal model of non-parameterised data types[1)]

I.1 Many-sorted algebras

A **many-sorted algebra** A is a 2-tuple <D,O> where

a) D is a set of sets, and the elements of D are referred to as the **data carriers** (of A); the elements of a data-carrier **dc** are referred to as **data-values**; and

b) O is a set of total functions, where the domain of each function is a Cartesian product of data carriers of A and the range of one of the data carriers.

I.2 Semantics of data type definitions

I.2.1 General concepts

I.2.1.1 Signature

A **signature** SIG is a tuple <S,OP> where

a) S is a set of **sort-identifiers** (also referred to as sorts); and

b) OP is a set of **operators.**

An operator consists of an **operation-identifier** op, a list of (argument) sorts w with elements in S, and a (range) sort s∈ S. This is usually written as op:w→s. If w is equal to the empty list the op:w→s is called a **null-ary** operator or **constant symbol** of sort s.

I.2.1.2 Signature morphism

Let $SIG_1=<S_1,OP_1>$ and $SIG_2=<S_2,OP_2>$ be signatures. A **signature morphism** $g:SIG_1 \rightarrow SIG_2$ is a pair of mappings

$$g = <gs:S_1 \rightarrow S_2, gop:OP_1 \rightarrow OP_2>$$

such that for all $e\text{-}opid_1 = <opidf_1, <gs(e\text{-}sidf_1), \ldots, gs(e\text{-}sidf_k)>, gs(e\text{-}res), pos > \in OP_1$

$$gop(\ e\text{-}opid_1) = <opidf_2, <(e\text{-}sidf_1), \ldots, (e\text{-}sidf_k)>, (e\text{-}res), pos >$$

for some operation-identifier $opidf_2$.

---

1) The text of this appendix has been agreed between CCITT and ISO as a common formal description of the initial algebra model for abstract data types. As well as appearing in this recommendation this text (with appropriate terminology, typographical and numbering changes) also appears in ISO IS8807. §§I.1, I.2.1.1, I.2.1.2, I.2.1.3, I.2.1.4, I.2.1.5, I.2.1.6, I.3, I.4.1, I.4.2, I.4.3, I.4.4, I.4.5 and I.4.6 of this appendix appear in §§5.2, 7.2.2.1, 7.3.2.8, 7.2.2.2, 7.2.2.3, 7.2.2.4, 7.2.2.5, 4.7, 7.4.2.1, 7.4.2.2, 7.4.3, 7.4.3 and 7.4.4 of IS8807 respectively. The terminologies **sort-identifier, operator, variable-identifier, variable, algebraic specification SPEC** and **operations** of this appendix are replaced by **sort-variable, operation-variable, value-variable, value-variable, data presentation pres** and **functions** respectively in IS8807.

I.2.1.3 Terms

Let V be any set of variables and let <S, OP> be a signature. The sets TERM(OP,V,s) of **terms** of sort s∈ S with operators in OP and variables in V, are defined inductively by the following steps:

a) each variable x:s ∈ V is in TERM(OP,V,s);

b) each null-ary operator op ∈ OP with res(op)= s is in TERM(OP,V,s);

c) if the terms t_i of sort s_i are in TERM(OP,V,s_i) for i=1,...,n, then for each op∈ OP with arg(op)=< s_1, ...,s_n> and res(op)=s, op(t_1,...,t_n) is in TERM(OP,V,s).

If term t is an element of TERM(OP,V,s) then s is call the sort of t, denoted as sort(t).
The set TERM(OP,s) of **ground terms** of sort s ∈ S is defined as the set TERM(OP,{},s).

I.2.1.4 Equations

An **equation** of sort s with respect to a signature <S,OP> is a triple <V,L,R> where

a) V is a set of variable-identifiers; and

b) L,R ∈ T(OP,V,s); and

c) s ∈ S.

An equation e'=<{},L',R'> is a **ground instance** of an equation e=<V,L,R>, if L',R' can be obtained from L,R by for each variable v:s in V, replacing all occurrences of that variable in L,R by the same ground term with sort s.

The notation L=R is used for the ground instance <{},L,R> of an equation.

Note – Also an equation <V,L,R> may be written L=R if no semantical complications are thus introduced.

I.2.1.5 Conditional equations

A **conditional equation** of sort s with respect to the signature <S, OP> is a triple <V,Eq,e>, where

a) V is a set of variable-identifiers; and

b) Eq is a set of equations with respect to <S, OP>, with variables in V; and

c) e is an equation of sort s with respect to <S, OP>, with variables in V.

I.2.1.6 Algebraic specifications

An **algebraic specification** SPEC is a triple <S,OP,E> where

a) <S,OP> is a signature; and

b) E is a set of conditional equations with respect to <S,OP>.

I.3 Derivation systems

A **derivation system** is a 3-tuple D=<A,Ax,I> with:

a) A a set, the elements of which are called **assertions,**

b) A⊇Ax the set of **axioms,**

c) I a set of **inference rules.**

Each inference rule R∈ I has the following format

$$R: \quad \frac{P_1,\ldots,P_n}{Q}$$

where $P_1,\ldots,P_n,Q \in A$.

A **derivation** of an assertion P in a derivation system D is a finite sequence s of assertions satisfying the following conditions:

a) the last element of s is P,

b) if Q is an element of s, then either Q∈ Ax, or there exists a rule R∈ I

$$R: \quad \frac{P_1,\ldots,P_n}{Q}$$

with $P_1,\ldots,P_n$ elements of s preceding Q.

If there exists a derivation of P in a derivation system D, this is written D ⊢P. If D is uniquely determined by context this may be abbreviated to ⊢P.

I.4 Semantics of algebraic specifications

All occurrences of a set of sorts S, a set operations OP, and a set of equations E in § I.4 refer to a given algebraic specification SPEC=<S,OP,E> as defined in § I.2.1.6.

In order to define the semantics of an algebraic specification SPEC, a derivation system associated with SPEC is used. This derivation system is defined in §§ I.4.1-I.4.3. Using this derivation system a relation on the set of ground terms with respect to <S,OP,E> and conguence classes are defined in § I.4.4 and § I.4.5. This relation is used in § I.4.6 to define an algebra (see § I.1) that represents the data type that is specified by <S,OP,E>.

I.4.1 Axioms generated by equations

Let ceq be a conditional equation. The set of axioms generated by ceq, notation Ax(ceq), is defined as follows:

a) if ceq=<V,Eq,e> with Eq≠{ }, then Ax(ceq)={ }; and

b) if ceq=<V,{ },e> then Ax(ceq) is the set of all ground instances of e (see § I.2.1.3)

I.4.2 Inference rules generated by equations

Let ceq be a conditional equation. The set of inference rules generated by ceq, notation Inf(ceq), is defined as follows:

a) if ceq=<V,{},e>, then Inf(ceq)={}, and

b) if ceq=<V, $\{e_1,\ldots,e_n\}$, e> with n>0, then Inf(ceq) contains all rules of the form

$$\frac{e_1',\ldots, e_n'}{e'}$$

where $e_1',\ldots, e_n',e'$ are ground instances of $e_1,\ldots, e_n,e$ respectively, that are obtained by, for each variable x occurring in V, replacing all occurrences of that variable in $e_1,\ldots, e_n,e$ by the same ground term with sort **sort(x)**.

I.4.3 Generated derivation system

The **derivation system** D=<A,Ax,I> (see § I.3) generated by an algebraic specification SPEC=<S,OP,E>is defined as follows:

a) A is the set of all ground instances of equations w.r.t. <S,OP>; and

b) Ax=$\bigcup${Ax(ceq) | ceq$\in$ E} $\cup$ ID,
with ID={t=t | t is a ground term}; and

c) I=$\bigcup${Inf(ceq) | ceq$\in$ E} $\cup$ SI,
where SI is given by the following schemata

i) $\dfrac{t_1=t_2}{t_2=t_1}$ for all ground terms t_1, t_2; and

ii) $\dfrac{t_1=t_2\ ,\ t_2=t_3}{t_1=t_3}$ for all ground terms t_1, t_2, t_3; and

iii) $$\frac{t_1=t_1',\ldots,\ t_n=t_n'}{op(t_1,\ldots,t_n)= op(t_1',\ldots,t_n')}$$
for all operators $op:s_1,\ldots,s_n\rightarrow s \in$ OP with n>0 and all ground terms of t_i,t_i' of sort s_i for i=1,...,n.

I.4.4 Congruence relation generated by an algebraic specification

Let D be the derivation system generated by an algebraic specification SPEC=<S,OP,E>. Two ground terms t_1 and t_2 are called **congruent** with respect to SPEC, notation $t_1\equiv_{SPEC}t_2$, iff

$$D \vdash t_1=t_2.$$

I.4.5 Congruence classes

The **SPEC-congruence class** [t] of a ground term t is the set of all terms congruent to t with respect to SPEC, i.e.

$$[\,t\,] = \{\, t' \mid t \equiv_{SPEC} t' \,\}.$$

I.4.6 Quotient term algebra

The semantical interpretation of an algebraic specification SPEC=<S,OP,E> is the following many-sorted algebra Q=<Dq,Oq>, called the **quotient term algebra**, where

a) Dq is the set { Q(s) | s∈ S} where
Q(s)= { [t] | t is ground term of sort s} for each s∈ S; and

b) Oq is the set of operations { op' | op∈ OP }, where the op' are defined by
op'([t_1],..., [t_n]) = [op(t_1,...,t_n)].

Annexes A, B, C and E
to Recommendation Z.100

FUNCTIONAL SPECIFICATION AND DESCRIPTION LANGUAGE (SDL)

PAGE INTENTIONALLY LEFT BLANK

PAGE LAISSEE EN BLANC INTENTIONNELLEMENT

ANNEX A

(to Recommendation Z.100)

SDL Glossary

The Z.100 Recommendation contains the formal definitions of SDL terminology. The SDL Glossary is compiled to help new SDL users when reading the Recommendation and its annexes, giving a brief definition and reference to the defining section of the Recommendation. The definitions in the Glossary may summarize or paraphrase the formal definitions, and thus may be incomplete.

Terms which are in a definition may also be found in the glossary. If an italicized phrase, for example *procedure identifier*, is not in the glossary, then it may be the concatenation of two terms, in this case the term *procedure* followed by the term *identifier*. When a word is in italics but cannot be located in the glossary, it may be a derivative of a glossary term. For example, *exported* is the past tense of *export*.

Except where a term is a synonym for another term, after the definition of the term there is a main reference to the use of the term in the Z.100 Recommendation. These references are shown in square brackets [ ] after definitions. For example, [3.2] indicates that the main reference is in § 3.2.

abstract data type

F : type abstrait de données

S : tipo abstracto de datos

Abstract data type is a synonym for *data type*. All *SDL data types* are *abstract data types*.

abstract grammar

F : grammaire abstraite

S : gramática abstracta

The *abstract grammar defines the semantics* of *SDL*. The *abstract grammer* is described by the *abstract syntax* and the *well-formedness rules*. [1.2, 1.4.1]

abstract syntax

F : syntaxe abstraite

S : sintaxis abstracta

The *abstract syntax* is the means to describe the conceptual structure of an *SDL specification* as compared with the *concrete syntaxes* which exist for each *concrete syntax* of *SDL*, this is *SDL/GR* and *SDL/PR*. [1.2]

access

F : accès

S : acceder

Access is the operation applied to a *variable* which gives the *value* which was last assigned to it. If a *variable* is *accessed* which has an *undefined value*, then an *error* occurs.

action

F : action

S : acción

An *action* is an operation which is executed within a *transition string*, e.g., a *task, output, decision, create request* or *procedure call*. [2.7]

active timer

F : temporisateur actif

S : temporizador activo

An *active timer* is a *timer* which has a *timer signal* in the *input port* of the owning *procedure* or is scheduled to produce a *timer signal* at some future time. [2.8.2, 5.5.4.5]

actual parameter

F : paramètre réel

S : paràmetro efectivo

An *actual parameter* is an *expression* given to a *process* or *procedure* for the corresponding *formal parameter* when the *process* or *procedure* is *created* (or *called*). Note that in certain cases in a procedure call an *actual parameter* must be a *variable* (i.e. a particular type of *expression*; see *IN/OUT*). [2.7.2, 2.7.3, 4.2.2]

actual parameter list

F : liste de paramètres réels

S : lista de paràmetros efectivos

An *actual parameter list* is the list of *actual parameters.* The *actual parameters* are matched by position with the respective elements of the corresponding *formal parameter list.*

area

F : zone

S : área; zona

An *area* is a two dimensional region in the *concrete graphical syntax. Area* often correspond to *nodes* in the *abstract syntax* and usually contain *common textual syntax.* In *interaction diagrams* areas may be connected by *channels* or *signal routes.* In *control flow diagrams areas* may be connected by *flow lines.*

array

F : tableau (array)

S : matriz

Array is the predefined *generator* used to introduce the concept of arrays, easing the definition of arrays.

assign

F : affectation

S : asignar

Assign is the operation applied to a *variable* which associates a *value* to the *variable* replacing the previous *value* associated with the *variable.* [5.5.3]

assignment statement

F : instruction d'affectation

S : sentencia de asignación

An *assignment statement* is a *statement* which assigns a *value* to a *variable.* [5.5.3]

association area

F : zone d'association

S : área de asociación

An *association area* is a connection between *areas* in an *interaction diagram* by means of an association *symbol.* There are five asociation areas: *channel substraction association area, input association area,* priority *input association area, continuous signal association area* and *save association area.* [2.6.3, 3.2.3, 4.10.2, 4.11]

axiom

F : axiome

S : axioma

An *axiom* is a special kind of *equation* with an implied equivalence to the *Boolean literal* True. "*Axioms*" is used as a synonym for "*axioms* and *equations.*" [5.1.3]

basic SDL

F : LDS de base

S : LED básico

Basic SDL is the subset of *SDL* defined in § 2 of Recommendation Z.100.

behaviour

F : comportement

S : comportamiento

The *behaviour* or *functional behaviour* of a *system* is the set of sequences of responses to sequences of stimuli. [1.1.3]

block

F : bloc

S : bloque

A *block* is part of a *system* or parent *block*. When used by itself, *block* is a synonym for a *block instance*. A *block* is a *scope unit* and provides a static interface. [2.4.3]

block area

F : zone de bloc

S : área de bloque

The *block area* is the definition of a *block* or a reference to a *block* in an *interaction diagram*. [2.4.2]

block definition

F : définition de bloc

S : definición de bloque

A *block definition* is the definition of a *block* in *SDL/PR*. [2.4.2]

block diagram

F : diagramme de bloc

S : diagrama de bloque

The *block diagram* is the definition of a *block* in *SDL/GR*. [2.4.3]

block substructure

F : sous-structure de bloc

S : subestructura de bloque

A *block substructure* is the *partitioning* of the *block* into *subblocks* and new *channels* at a lower *level of abstraction*. [3.2.2]

block substructure definition

F : définition de sous-structure de bloc

S : definición de subestructura de bloque

A *block substructure definition* is the *SDL/PR* representation of a *block substructure* for a partitioned block. [3.2.2]

block substructure diagram

F : diagramme de sous-structure de bloc

S : diagrama de subestructura de bloque

A *block substructure diagram* is the *SLD/GR* representation of a *block substructure* for a *partitioned block.* [3.2.2]

block tree diagram

F : diagramme d'arborescence de bloc

S : diagrama de árbol de bloques

A *block tree diagram* is an auxiliary document in *SDL/GR* representing the *partitioning* of a *system* into *blocks* at lower *levels of abstraction* by means of an inverted tree diagram (*i.e.*, parent *block* at the top). [3.22]

BNF (Backus-Naur Form)

F : forme BNF (Backus-Naur Form)

S : FBN (forma Backus-Naur)

BNF (Backus-Naur Form) is a formal notation used for expressing the *concrete textual syntax* of a language. An extended form of *BNF* is used for expressing the *concrete graphical grammar.* [1.5.2, 1.5.3]

Boolean

F : booléen

S : booleano

Boolean is a *sort* defined in a predefined *partial type definition* and has the *values* True and False. For the *sort Boolean* the predefined *operators* are NOT, AND, OR, XOR and implication. [5.6.1]

channel

F : canal

S : canal

A *channel* is the connection conveying *signals* between two *blocks. Channels* also convey *signals* between a *block* and the *environment. Channels* may be unidirectional or bidirectional. [2.5.1]

channel definition

F : définition

S : definición de canal

A *channel definition* is the definition of a *channel* in *SDL/PR.* [2.5.1]

channel definition area

F : zone de définition de canal

S : área de definición de canal

The *channel definition area* is the definition of a *channel* in *SDL/GR.* [2.5.1]

channel substructure

F : sous-structure de canal

S : subestructura de canal

A *channel substructure* is a *partitioning* of a *channel* into a set of *channels* and *blocks* at a *lower level of abstraction.* [3.2.3]

channel substructure definition

F : définition de sous-structure de canal

S : definición de subestructura de canal

A *channel substructure definition* is the definition of the *channel substructure* in *SDL/PR.* [3.2.3]

channel substructure diagram

F : diagramme de sous-structure de canal

S : diagrama de subestructura de canal

A *channel substructure diagram* is the definition of the channel substructure in *SDL/GR.* [3.2.3]

character

F : caractère (character)

S : carácter; character

Character is a *sort* defined in a predefined *partial type definition* for which the *values* are the elements of the CCITT No. 5 alphabet, (e.g., 1, A, B, C, etc.). For the *sort character* the *ordering operators* are predefined. [5.6.2]

chartstring

F : chaîne de caractères (character string)

S : cadena-de-caracteres; chartstring

Chartstring is a *sort* defined in a predefined *partial type definition* for which the *values* are *strings* of *characters* and the *operators* are those of the *string* predefined *generator* instantiated for *characters.* [5.6.4]

comment

F : commentaire

S : comentario

A *comment* is information which is in addition to or clarifies the *SDL specification.* In *SDL/GR comments* may be attached by a dashed line to any *symbol.* In *SDL/PR comments* are introduced by the keyword *COMMENT. Comments* have no *SDL* defined meaning. See also *Note.* [2.2.6]

common textual grammar

F : grammaire textuelle commune

S : gramática textual común

The *common textual grammar* is the subset of the *concrete textual grammar* which applies to both *SDL/GR* and *SDL/PR.* [1.2]

communication path

F : trajet de communication

S : trayecto de comunicación

A *communication path* is a transportation means that carriers *signal instances* from one *process instance* or from the *environment* to another *process instance* or to the *environment.* A *communication path* comprises either *channel* path(s) or *signal route* path(s) or a combination of both. [2.7.4]

complete valid input signal set

F : ensemble complet de signaux d'entrée valides

S : conjunto completo de señales de entrada válidas

The *complete valid input signal set* of a *process* is the union of the *valid input signal* set, the local *signals, timer signals* and the *implicit signals* of the *process.* [2.4.4]

concrete grammar

F : grammaire concrète

S : gramática concreta

A *concrete grammar* is the *concrete syntax* along with the *well-formedness rules* for that *concrete syntax. SDL/GR* and *SDL/PR* are the *concrete grammars* of *SDL.* The *concrete grammars* are mapped to the *abstract grammar* to determine their *semantics.* [1.2]

concrete graphical grammar

F : grammaire graphic concréte

S : gramática gráfica concreta

The *concrete graphical grammar* is the *concrete grammar* for the graphical part of SDL/GR.

concrete graphical syntax

F : syntaxe graphique concrète

S : sintaxis gráfica concreta

The *concrete graphical syntax* is the *concrete syntax* for the graphical part of *SDL/GR.* The *concrete graphical syntax* is expressed in Z.100 using an extended form of *BNF.* [1.2, 1.5.3]

concrete syntax

F : syntaxe concrète

S : sintaxis concreta

The *concrete syntax* for the various representations of *SDL* is the actual *symbols* used to represent *SDL* and the interrelationship between *symbols* required by the syntactic rules of *SDL.* The two *concrete syntaxes* used in Z.100 are the *concrete graphical syntax* and the *concrete textual syntax.* [1.2]

concrete textual syntax

F : syntaxe textuelle concrète

S : sintaxis textual concreta

The *concrete textual syntax* is the *concrete syntax* for *SDL/PR* and the textual parts of *SDL/GR.* The *concrete textual syntax* is expressed in Z.100 using *BNF.* [1.2, 1.5.2]

conditional expression

F : expression conditionnelle

S : expresión condicional

A *conditional expression* is an *expression* containing a *Boolean expression* which controls whether the consequence *expression* or the alternative *expression* is interpreted. [5.5.2.3]

connect

F : connect

S : conectar

Connect indicates the connection of a *channel* to one or more *signal routes.* [2.5.3]

connector

F : connecteur

S : conector

A *connector* is an *SDL/GR symbol* which is either an *in-connector* or an *out-connector.* A *flow line* is implied from *out-connectors* to the associated *in-connector* in the same *process* or *procedure* identified by having the same *name.* [2.6.6]

consistent partitioning subset

F : sous-ensemble de subdivision cohérent

S : subconjunto de partición consistente

A *consistent partitioning subset* is a set of the *blocks* and *subblocks* in a *system specification* which provides a complete view of the *system* with related parts at a corresponding *level of abstraction.* Thus, when a *block* or *subblock* is contained in a *consistent partitioning subset,* its ancestors and siblings are too. [3.2.1]

consistent refinement subset

F : sous-ensemble de raffinement cohérent

S : subconjunto de refinamiento consistente

The *consistent refinement subset* is a *consistent partitioning subset* which contains all *blocks* and *subblocks* which use the *signals* used by any of the *blocks* or *subblocks.* [3.3]

continuous signal

F : signal continu

S : señal continua

A *continuous signal* is a means to define that when in a *state* the associated *Boolean* condition becomes True, the *transition* following the *continuous signal* is interpreted. [4.11]

control flow diagram

F : diagramme de liaison de contrôle

S : diagrama de flujo de control

A *control flow diagram* is either a *process diagram,* a *procedure diagram,* or a *service diagram.*

create

F : créer

S : crear

Create is a synonym for *create request.*

create request

F : demande de création

S : petición de crear

A *create request* is the *action* causing the creation and starting of a new *process instance* using a specified *process type* as a template. The *actual parameters* in the *create request* replace the *formal parameters* in the *process.* [2.7.2]

create line area

F : zone de ligne de création

S : área de línea de crear

The *create line area* in a *block diagram* connects the *process area* of the *creating (PARENT) process* with the *process area* of the *created (OFFSPRING) process.*[2.4.3]

data type

F : type de données

S : tipo de datos

A *data type* is the definition of sets of *values (sorts),* a set of *operators* which are applied to these *values* and a set of algebraic rules (*equations*) defining the *behaviour* when the *operators* are applied to the *values.* [2.3.1]

data type definition

F : définition de type de données

S : definición de tipo de datos

A *data type definition* defines the validity of *expressions* and relationship between *expressions* at any given point in an *SDL specification.* [5.2.1]

decision

F : décicion

S : decisión

A *decision* is an *action* within a *transition* which asks a question to which the answer can be obtained at that instant and accordingly chooses one of the several outgoing *transitions* from the *decision* to continue interpretation. [2.7.5]

decision area

F : zone de décision

S : área de decisión

A *decision area* is the *SDL/GR* representation of a *decision.* [2.7.5]

default

F : défault

S : por defecto

The *default assignment* is a denotation of a *value* that is initially associated to each *variable* of the *sort* of the *default* clause. The *default* clause may appear in *data type definitions.* [5.5.3.3]

description

F : description

S : descripción

A *description* of a *system* is the description of its actual *behaviour.* [1.1]

diagram

F : diagramme

S : diagrama

A *diagram* is the *SDL/GR* representation for a part of a *specification.* [2.4.2]

duration

F : durée (duration)

S : duración; duration

Duration is a *sort* defined in a predefined *partial type definition* for which the *values* are denoted as *reals* and represent the interval between two time instants. [5.6.11]

enabling condition

F : condition de validation

S : condición habilitante (o habilitadora)

An *enabling condition* is a means for conditionally accepting a *signal* for *input.* [4.12]

enabling condition area

F : zone de condition de validation

S : área de condición habilitante (o habilitadora)

The *enabling condition area* is the *SDL/GR* representation of an *enabling condition.* [4.12]

entity class

F : classe d'entité

S : clase de entidad

An *entity class* is a categorization of *SDL types* based on similarity of use. [2.2.2]

environment

F : environnement

S : entorno

The term *environment* is a synonym for the *environment of a system.* Also when context allows, it may be a synonym for the *environment* of a *block, process, procedure* or a *service.* [1.3.2]

environment of a system

F : environnement d'un système

S : entorno de un sistema

The *environment of a system* is the external world of the *system* being specified. The environment interacts with the *system* by sending/receiving *signal instances* to/from the *system.* [1.3.2]

equation

F : équation

S : ecuación

An *equation* is a relation between *terms* of the same *sort* which holds for all possible *values* substituted for each *value identifier* in the *equation.* An *equation* may be an *axiom.* [5.1.3, 5.2.3]

error

F : erreur

S : error

An *error* occurs during the interpretation of a *valid specification* of a *system* when one of the dynamic conditions *SDL* is violated. Once an *error* has occurred, the subsequent *behaviour* of the *system* is not defined by *SDL.* [1.3.3]

export

F : export

S : exportación

The term *export* is a synonym for *export operation.*

exported variable

F : variable exportée

S : variable exportada

An *exported variable* is a *variable* which can be used in an *export operation.* [4.13]

exporter

F : exportateur

S : exportador

An *exporter* of a *variable* in the *process instance* which owns the *variable* and *exports* its *values.* [4.13]

export operation

F : opération d'exportation

S : operación de exportación

An *export operation* is the operation by which the *exporter* discloses the *value* of a *variable.* See *import operation.* [4.13]

expression

F : expression

S : expresión

An *expression* is either a *literal,* an *operator* application, a *synonym,* a *variable access,* a *conditional expression,* or an *imperartive operator* applied to one or more *expressions.* When an *expression* is interpreted a *value* is obtained (or the *system* is in *error*). [2.3.4, 5.4.2.1]

external synonym

F : synonyme externe

S : sinónimo externo

An *external synonym* of a predefined *sort* whose *value* is not specified in the *system specification.* [4.3.1]

extract!

F : extract!

S : extraer!; extract!

Extract is an *operator* which is implied in an *expression* when a *variable* is immediately followed by bracketed *expression(s).* [5.4.2.4, 5.6.8]

flow line

F : ligne de liaison

S : línea de flujo

A *flow line* is a *symbol* used to connect *areas* in a *control flow diagram.* [2.2.4, 2.6.7.2.2]

formal parameter

F : paramètre formel

S : parámetro formal

A *formal parameter* is a *variable name* to which *actual values* are assigned or which are replaced by *actual variables.* [2.4.4, 2.4.5, 4.2, 4.10]

formal parameter list

F : liste de paramètres formels

S : lista de parámetros formales

A *formal parameter list* is list of a *formal parameters.*

functional behaviour

F : comportement fonctionnel

S : comportamiento funcional

Functional behaviour is a synonym for *behaviour.*

general option area

F: zone d'option générale

S: área de opción general

The *general option area* is the *SDL/GR* representation of an *option.* [4.3.3]

general parameters

F : paramètres généraux

S : parámetros generales

The *general parameters* in both a *specification* and a *description* of a *system* relate to such matters as temperature limits, construction, exchange capacity, grade of service, etc., and are not defined in *SDL.* [1.1]

generator

F : générateur

S : generador

A *generator* is an incomplete *newtype* description. Before it assumes the status of a *newtype,* a *generator* must be instantiated by providing the missing information. [5.4.1.1.2]

graph

F : graphe

S : gráfico

A *graph* in the *abstract syntax* is a part of an *SDL specification* such as *procedure graph* or a *process graph.*

ground expression

F : expression close

S : expresión fundamental

A *ground expression* is an *expression* containing only *operators, synonyms* and *literals.* [5.4.2.2]

hierarchical structure

F : structure hiérarchique

S : estructure jeràrquica

A *hierarchical structure* is a structure of a *system specification* where *partitioning* and *refinement* allow different views of the *system* at different *levels of abstraction. Hierarchical structures* allow the management of complex *system specifications.* See also *block tree diagram.* [3.1]

identifier

F : identificateur

S : identificador

An *identifier* is the unique identification of an object, formed from a *qualifier part* and a *name.* [2.2.2]

imperative operator

F : opérateur impératif

S : operador imperativo

An *imperative operator* is a now *expression, view expression, timer active expression, import expression* or one of the *PId expressions: SELF, PARENT, OFFSPRING* or *SENDER.* [5.5.4]

implicit transition

F : transition implicite

S : transición implícita

An *implicit transition* is in the *concrete syntax* initiated by a *signal* in the *complete valid input signal set* and not specified in an *input* or *save* for the *state.* An *implicit transition* contains no *action* and leads directly back to the same *state* [4.6]

import

F : import

S : importación

The term *import* is a synonym for *import operation.* [4.13]

imported variable

F : variable importée

S : variable importada

An *imported variable* is a *variable* used in an *import operation.* [4.13]

importer

F : importeur

S : importador

An *importer* of an *imported variable* is the *process instance* which *imports* the *value.* [4.13]

import operation

F : opération d'importation

S : operación de importación

An *import operation* is the operation that yields *value* of an *exported variable.* [4.13]

IN variable

F : variable "IN"

S : variable IN

An *IN variable* is a *formal parameter* attribute denoting the case when a *value* is passed to a *procedure* via an *actual parameter.* [2.4.5]

IN/OUT variable

F : variable "IN/OUT"

S : variable IN/OUT

An *IN/OUT variable* is a *formal parameter* attribute denoting the case when a *formal parameter name* is used as a synonym for the *variable* (i.e. the *actual parameter* must be a *variable.* [2.4.5]

in-connector

F : connecteur d'entrée

S : conector de entrada

An *in-connector* is a *connector.*

infix operator

F : opérateur infixe

S : operador infijo

An *infix operator* is one of the predefined dyadic *operators* of *SDL* (=>, OR, XOR, AND, IN, /=, =, >, <, <=, >=, +, −, //, *, /, MOD, REM) which are placed between its two arguments. [5.4.1.1]

informal text

F : texte informel

S : texto informal

Informal text is text included in an *SDL specification* for which *semantics* are not defined by *SDL*, but through some other model. *Informal text* is enclosed in apostrophes. [2.2.3]

initial algebra

F : algèbre initiale

S : álgebra inicial

An *initial algebra* is the formalism for defining *abstract data types.* [5.3]

inlet

F : accès entrant

S : acceso de entrada

An *inlet* represents a line, such as a *channel* or a *flow line*, entering an *SDL/GR macro call.* [4.2.3]

input

F : entrée

S : entrada

An *input* is the consumption of a *signal* from the *input port* which starts a *transition.* During the consumption of a *signal*, the *values* associated with the *signal* become available to the *process instance.* [2.6.4, 4.10.2]

input area

F : zone d'entrée

S : área de entrada

An *input area* is the *SDL/GR* representation of an *input.* [2.6.4]

input port

F : port d'entrée

S : puerto de entrada

An *input port* of a *process* is a queue which receives and retains *signals* in the order of arrival until the *signals* are consumed by an *input.* The *input port* may contain any number of *retained signals.* [2.4.4]

instance

F : instance

S : instancia

An *instance* of a *type* is an object which has the properties of the *type* (given in the definition). [1.3.1]

instantiation

F : instantiation

S : instanciación

Instantiation is the creation of an *instance* of a *type.* [1.3.1]

integer

F : entier (integer)

S : entero; integer

Integer is a *sort* defined in a predefined *partial type definition* for which the *values* are these of mathematical integers (. . ., −2, −1, 0, +1, +2, . . .). For the *sort integer* the predefined *operators* are +, −, *, / and the *ordering operators.* [5.6.5]

interaction diagram

F : diagramme d'interaction

S : diagrama de interacción

An *interaction diagram* is a *block diagram, system diagram, channel substructure diagram,* or *block substructure diagram.*

keyword

F : mot clé

S : palabra clave

A *keyword* is a reserved *lexical unit* in the *concrete textual syntax.* [2.2.1]

label

F : étiquette

S : etiqueta

A *label* is a *name* followed by a colon and is used in the *concrete textual syntax* for connection purposes. [2.6.6]

level

F : niveau

S : nivel

The term *level* is a synonym for *level of abstraction.*

level of abstraction

F : niveau d'abstraction

S : nivel de abstracción

A *level of abstraction* is one of the levels of a *block tree diagram.* A description of a *system* is one *block* at the highest *level of abstraction* and is shown as a single *block* at the top of a *block tree diagram.* [3.2.1]

lexical rules

F : règles lexicales

S : reglas léxicas

Lexical rules are rules which define how *lexical units* are built from characters. [2.2.1, 4.2.1]

lexical unit

F : unités lexicales

S : unidad léxica

Lexical units are the terminal *symbols* of the *concrete textual syntax.* [2.2.1]

literal

F : littéral

S : literal

A *literal* denotes a *value.* [2.3.3, 5.1.2, 5.4.1.14]

macro

F : macro

S : marco

A *marcro* is a named collection of syntactic or textual items, which replaces the *macro call* before the meaning of the *SDL* representation is considered (i.e., a *macro* has meaning only when replaced in a particular context). [4.2]

macro call

F : appel de macro

S : llamada a (de) macro

A *macro call* is an indication of a place where the *macro definition* with the same *name* should be expanded. [4.2.3]

macro definition

F : définition de macro

S : definición de macro

A *macro definition* is the definition of a *macro* in *SDL/PR.* [4.2.2]

macro diagram

F : diagramme de macro

S : diagrama de macro

A *macro diagram* is the definition of a *macro* in *SDL/GR.* [4.2.2]

make!

F : make!

S : hacer!; make!

Make! is an operation only used in *data type* definitions fo form a *value* of a complex type (e.g., *structured sort*). [5.4.1.10, 5.6.8]

merge area

F : zone de fusion

S : área de fusión

A *merge area* is where one *flow line* connects to another. [2.6.7.2.2]

Meta IV

F : Meta IV

S : Meta IV

Meta IV is a formal notation for expressing the *abstract syntax* of a language. [1.5.1]

model

F : modèle

S : modelo

A *model* gives the mapping for *shorthand* notations expressed in terms of previously defined *concrete syntax.* [1.4.1, 1.4.2]

modify!

F : modify!

S : modificar!; modify!

Modify is an *operator* which is implied in *expressions* when a *variable* is immediately followed by bracketed expressions and then :=. Within axioms *modify!* is used explicitly (see *extract!*) [5.4.1.10, 5.6.8]

name

F : nom

S : nombre

A *name* is a *lexical unit* used to name *SDL* objects. [2.2.1, 2.2.2]

natural

F : naturel

S : natural

Natural is a *syntype* defined in a predefined *partial type definition* for which the *values* are the non-negative integers (i.e., 0, 1, 2, ...). The *operators* are the *operators* of the *sort integer.* [5.6.6]

newtype

F : nouveau type (newtype)

S : niotipo

A *newtype* introduces a *sort,* a set of *operators,* and a set of *equations.* Note that the term *newtype* might be confusing because actually a new *sort* is introduced, but *newtype* is maintained for historical reasons. [5.2.1]

node

F : noeud

S : nodo

In the *abstract syntax,* a *node* is a designation of one of the basic concepts of *SDL*.

note

F: note

S: nota

A *note* is text enclosed by /* and */ which has no *SDL* defined semantics. See *comment.* [2.2.1]

null

F : null

S : null; nulo

Null is the *literal* of *sort PId.* [5.6.10]

OFFSPRING

F : DESCENDANT (OFFSPRING)

S : OFFSPRING; VASTAGO

OFFSPRING is an *expression* of *sort PId.* When *OFFSPRING* is evaluated in a *process* it gives the *PId-values* of the *process* most recently *created* by this *process.* If the *process* has not *created* any *processes,* the result of the evaluation of *OFFSPRING* is *null.* [2.4.4, 5.5.4.3]

operator

F : opérateur

S : operador

An *operator* is a denotation for an operation. *Operators* are defined in a *partial type definition.* For example +, −, *, /, are *names* for *operators* defined for *sort integer.* [5.1.2, 5.1.3]

operator signature

F : signature d'opérateur

S : signatura de operador

An *operator signature* defines the *sort(s)* of the *values* to which the *operator* can be applied and the *sort* of the resulting *value.* [5.2.2]

option

F : option

S : opción

An *option* is a *concrete syntax* construct in a generic *SDL system specification* allowing different *system* structures to be chosen before the *system* is interpreted. [4.3.3, 4.3.4]

ordering operators

F : opérateurs de relation d'ordre

S : operadores de ordenación

The *ordering operators* are <, <=, > or >=. [5.4.1.8]

out connector

F : connecteur de sortie

S : conector de salida

An *out-connector* is a connector.

outlet

F : accès sortant

S : acceso de salida

An *outlet* represents a line, such as a *channel* or *flow line*, existing a *macro diagram.* [4.2.2]

output

F: sortie

S: salida

An *output* is an *action* within a *transition* which generates a *signal instance.*

output area

F : zone de sortie

S : área de salida

The *output area* in a *control flow diagram* represents the *SDL/GR* concept of an *output.* [2.7.4]

page

F : page

S : página

A *page* is one of the components of a physical partitioning of a *diagram.* [2.2.5]

PARENT

F : PARENT

S : PARENT; PROGENITOR

PARENT is a *PId expression.* When a *process* evaluates this *expression,* the result is the *PId-value* of the parent *process.* If the *process* was created at *system* initialization time, the result is *null.* [2.4.4, 5.5.4.3]

partial type definition

F : défintiion partielle de type

S : definición parcial de tipo

The *partial type definition* for a *sort* defines some of the properties related to the *sort.* A *partial type definition* is part of a *data type* definition. [5.2.1]

partitioning

F : subdivision

S : partición

Partitioning is the subdivision of a unit into smaller components which when taken as a whole have the same *behaviour* as the original unit. *Partitioning* does not affect the static interface of a unit. [3.1, 3.2]

PId

F : PId

S : PId

PId is a *sort* defined in a predefined *partial type definition* for which there is one *literal, null. PId* is an abbreviation for process instance identifier, and the *values* of the *sorts* are used to identify *process instances.* [5.5.4.3, 5.6.10]

powerset

F : mode ensembliste

S : conjunista

Powerset is the *predefined generator* used to introduce mathematical sets. The *operators* for *powerset* are IN, Incl, Del, union, insersection and the *ordering operators.* [5.6.9]

predefined data

F : données prédéfinies

S : datos predefinidos

For simplicity of description the term *predefined data* is applied to both predefined *names* for *sorts* introduced by *partial type definitions* and predefined *names* for *data type generators. Boolean, character, chartstring, duration, integer, natural PId, real* and *time* are *sort names* which are predefined. *Array, powerset,* and *string* are *data type generator names*which are predefined. *Predefined data* are defined implicitly at *system level* in all *SDL systems.* [5.6]

procedure

F : procédure

S : procedimiento

A *procedure* is an encapsulation of the *behaviour* of a *process.* A *procedure* is defined in one place but may be referred to several times within the same *process.* See *formal parameter* and *actual parameter.* [2.4.5]

procedure call

F : appel de procédure

S : llamada a (de) procedimiento

A *procedure call* is the invocation of a named *procedure* for interpretation of the *procedure* and passing *actual parameters* to the *procedure.* [2.7.3]

procedure call area

F : zone d'appel de procédure

S : área de llamada a (de) procedimiento

The *procedure call area* is the *SDL/GR* representation of a *procedure call.* [2.7.3]

procedure definition

F : définition de procédure

S : definición de procedimiento

A *procedure definition* is the *SDL/PR* definition of a *procedure.* [2.4.5]

procedure diagram

F : diagramme de procédure

S : diagrama de procedimiento

A *procedure diagram* is the *SDL/GR* representation of a *procedure.* [2.4.5]

procedure graph

F : graphe de procédure

S : gráfico de procedimiento

A *procedure graph* is a nonterminal in the *abstract syntax* representing a *procedure.* [2.4.5]

procedure return

F : retour de procédure

S : retorno de procedimiento

Procedure return is a synonym for *return.*

process

F : processus

S : proceso

A *process* is a communicating extended finite state machine. Communication can take place via *signals* or shared *variables.* The *behaviour* of a *process* depends on the order of arrival of *signals* in its *input port.* [2.4.4]

process area

F : zone de processus

S : área de proceso

A *process area* in *SDL/GR* is the representation of a *process* or a reference to a *process* in an *interaction diagram.* [2.4.3]

process definition

F : définition de processus

S : definición de processo

A *process definition* is the *SDL/PR* representation of a *process.* [2.4.4]

process diagram

F : diagramme de processus

S : diagrama de proceso

A *process diagram* is the *SDL/GR* representation of the definition of a *process.* [2.4.4]

process graph

F : graphe de processus

S : gráfico de proceso

A *process graph* is nonterminal in the *abstract syntax* representing a *process.* [2.4.4]

process instance

F : instance de processus

S : instancia de proceso

A *process instance* is a dynamically created *instance* ofa *process.* See *SELF, SENDER, PARENT,* and *OFFSPRING* [2.4.4]

qualifier

F : partie qualificative (qualificatif)

S : calificador

The *qualifier* is part of an *identifier* which is the extra information to the *name* part of the *identifier* to ensure uniqueness. *Qualifiers* are always present in the *abstract syntax,* but only have to be used as far as needed for uniqueness in the *concrete syntax* when the *qualifier* of an *identifier* cannot be derived from the context of the use of the *name* part. [2.2.2]

real

F : réel

S : real

Real is a *sort* defined in a predefined *partial type definition* for which the *values* are the numbers which can be presented by one *Integer* divided by another. The predefined *operators* for the *sort real* have the same *names* as the *operators* of *sort integer.* [5.6.7]

refinement

F : reaffinement

S : refinamiento

Refinement is the addition of new details to the funtionality at a certain *level of abstraction.* The *refinement* of a *system* causes an enrichment in its *behaviour* or its capabilities to handle more types of *signals* and information, including those *signals* to and from the *environment.* Compare with *partitioning.* [3.3]

remote definition

F : définition distante

S : definición remota

A *remote definition* is a syntactic means of distributing a *system definition* into several parts and relating the parts to each other. [2.4.1]

reset

F : reset (réinitialisation)

S : reincializar; reponer

Reset is an operation defined for *timers* which allows *timers* to be made inactive. See *active timer.* [2.8]

retained signal

F : signal retenu

S : señal retenida

A *retained signal* is a *signal* in the *input port* of a *process*, i.e., a *signal* which has been received but not consumed by the *process.* [2.4.4]

return

F : retour

S : retorno

The *return* of a *procedure* is the transfer of control to the calling *procedure* or *process.* [2.6.7.2.4]

reveal attribute

F : attribut d'exposition

S : atributo revelado

A *variable* owned by a *process* may have a *reveal attribute*, in which case another *process* in the same *block* is permitted to view the *value* associated with the *variable.* See *view definition.* [2.6.1.1]

save

F : mise en réserve

S : conservación

A *save* is the declaration of those *signals* that should not be consumed in a given *state.* [2.6.5]

save area

F : zone de mise en réserve

S : área de conservación

The *save area* is the *SDL/GR* representation of a *save.* [2.6.5]

save signal set

F : ensemble de signaux de mise en réserve

S : conjunto de señales de conservación

The *save signal set* of a *state* is the set of saved *signals* for that *state.* [2.6.5]

SDL (CCITT Specification and Description Language)

F : LDS (langage de description et de spécification du CCITT)

S : LED (lenguaje de especificación y descripción del CCITT)

CCITT *SDL* (*Specification and Description Language*) is a formal language providing a set of constructs of the *specification* for the functionality of a system.

SDL/GR

F: LDS/GR

S: LED/GR

SDL/GR is the graphical representation in *SDL.* The *grammar* for *SDL/GR* is defined by the *concrete graphical grammar* and the *common textual grammar.* [1.2]

SDL/PE

F : LDS/PE

S : LED/EP

SDL/PE is a set of icons which can be used in conjunction with the *state symbol* of *SDL/GR.* [Annex E]

SDL/PR

F : LDS/PR

S : LED/PR

SDL/PR is the textual phrase representation in *SDL.* The *grammar* for *SDL/PR* is defined by the *concrete textual grammar.* [1.2]

scope unit

F : unité de portée

S : unidad de ámbito

A *scope unit* in the *concrete grammar* defines the range of *visibility* of *identifiers.* Examples of *scope units* include the *system, block, process, procedure, partial type definitions* and *service definitions.* [2.2.2]

selection

F : sélection

S : selección

Selection means providing those *external synonyms* needed to make a specific *system specification* from a generic *system specification.* [4.3.3]

SELF

F : SELF

S : SELF; MISMO

SELF is a *PId expression.* When a *process* evaluates this *expression,* the result is the *PId-value* of that *process. SELF* never results in the *value Null.* See also *PARENT, OFFSPRING, PId.* [2.4.4, 5.5.4.3]

semantics

F : sémantique

S : semántica

Semantics gives meaning to an entity: the properties it has, the way its *behaviour* is interpreted, and any dynamic conditions which must be fulfilled for the *behaviour* of the entity to meet *SDL* rules. [1.4.1, 1.4.2]

SENDER

F : SENDER (émetteur)

S : SENDER; EMISOR

SENDER is a *PId expression.* When evaluated *SENDER* yields the *PId value* of the sending *process* of the *signal* that activated the current *transition.* [2.4.4, 2.6.4, 5.5.4.3]

service

F : service

S : servicio

A *service* is an alternative way of specifying a *process.* Each *service* may define a partial *behaviour* of a *process.* [4.10]

service area

F : zone de service

S : área de servicio

A *service area* is either a *service diagram* or a reference to a *service.* [4.10.1]

service definition

F : définition de service

S : definición de servicio

A *service definition* is the *SDL/PR* definition of a *service.* [4.10.1]

service diagram

F : diagramme de service

S : diagrama de servicio

A *service diagram* is the *SDL/GR* definition of a *service.* [4.10]

set

F : set (initialisation)

S : inicializar; poner

Set is an operation defined for *timers* which allow *timers* to be made *active.* [2.8]

shorthand notation

F : notation abrégée

S : notación taquigráfica (o abreviada)

A *shorthand notation* is a *concrete syntax* notation providing a more compact representation implicitly referring to *Basic SDL* concepts. [1.4.2]

signal

F : signal

S : señal

A *signal* is an instance of a signal *type* communication information to a *process instance.* [2.5.4]

signal definition

F : définition de signal

S : definición de señal

A *signal definition* defines a *named signal type* and associates a list of zero or more *sort identifiers* with the *signal name.* This allow *signals* to carry *values.* [2.5.4]

signal list

F : liste de signaux

S : lista de señales

A *signal list* is a list of *signal identifiers* used in *channel* and *signal route definitions* to indicate all the *signals* which may be conveyed by the *channel* or *signal route* in one direction. [2.5.5]

signal list area

F : zone de liste de signaux

S : área de lista de señales

The *signal list area* in an *interaction diagram* represents a *signal list* associated with a *channel* or *signal route.* [2.5.5]

signal route

F : acheminement de signaux

S : ruta de señales

A *signal route* indicates the flow of *signals* between a *process type* and either another *process type* in the same *block* or the *channels* connected to the *block.* [2.5.2]

simple expression

F : expression simple

S : expresión simple

A *simple expression* is an *expression* which only contains *operators, synonyms,* and *literals* of the predefined *sorts.* [4.3.2]

sort

F : sorte

S : género

A *sort* is a set of *values* with common characteristics. *Sorts* are always nonempty and disjoint. [2.3.3, 5.1.3]

specification

F : spécification

S : especificación

A *specification* is a definition of the requirements of a *system.* A *specification* consists of *general parameters* required of the *system* and the *functional specification* of its required *behaviour. Specification* may be also used as a shorthand for "*specification* and/or *description*", e.g., in *SDL specification* or *system specification.* [1.1]

start

F : départ

S : arranque

The *start* in a *process* is interpreted before any *state* or *action.* The *start* initializes the *process* by replacing its *formal parameters* by the *actual parameters* as specified in the *create.* [2.6.2]

state

F : état

S : estado

A *state* is a condition in which a *process instance* can consume a *signal.* [2.6.3]

state area

F : zone d'état

S : área de estado

A *state area* is the *SDL/GR* representation of one or more *states.* [2.6.3]

state picture

F : représentation graphique d'état

S : pictograma de estado

A *state picture* is a *state symbol* incorporating pictorial elements used to extend *SDL/GR* to *SDL/PE.* [Annex E]

stop

F : arrêt

S : parada

A *stop* is an action which terminates a *process instance.* When a *stop* is interpreted, all *variables* owned by the *process instance* are destroyed and all *retained signals* in the *input port* are no longer accessible. [2.6.7.2.3]

string

F : chaîne (string)

Ŝ: cadena; string

String is a predefined *generator* used to introduce lists. The predefined *operators* include Length, First, Last, Substring and concatenation. [5.6.3]

structured sort

F : sorte structurée

S : género estructurado

A *structured sort* is a *sort* with implicit *operators* and *equations* and special *concrete syntax* for these implicit *operators.* The *structured sort* is used to make *values* with so called fields. The *values* of the fields can be *accessed* and *modified* independently. [5.4.1.10]

subblock

F : sous-bloc

S : subbloque

A *subblock* is a *block* contained within another *block*. *Subblocks* are formed when a *block* is *partitioned*. [3.2.1, 3.2.2]

subchannel

F : sous-canal

S : subcanal

A *subchannel* is a *channel* formed when a *block* is *partitioned*. A *subchannel* connects a *subblock* to a boundary of the *partitioned block* or a *block* to the boundary of a *partitioned channel*. [3.2.2, 3.2.3]

subsignal

F : sous-signal

S : subseñal

A *subsignal* is a *refinement* of a *signal* and may be further *refined*. [3.3]

symbol

F : symbole

S : símbolo

A *symbol* is a terminal in the *concrete syntaxes*. A *symbol* may be one of a set of shapes in the *concrete graphical syntax*.

synonym

F : synonyme

S : sinónimo

A *synonym* is a *name* which represents a *value*. [5.4.1.13]

syntax diagram

F : diagramme de syntaxe

S : diagrama de sintaxis

Syntax diagrams are illustrations of the definitions of the *concrete textual syntax*. [Annex C2]

syntype

F : syntype

S : sintipo

A *syntype* determines a set of *values* which corresponds to a subset of the *values* of the parent *type*. The *operators* of the *syntype* are the same as those of the parent *type*. [5.4.1.9]

system

F : système

S : sistema

A *system* is a set of *blocks* connected to each other and the *environment* by *channels*.

system definition

F : définition de système

S : definición de sistema

A *system definition* is the *SDL/PR* representation of a *system.* [2.4.2]

system diagram

F : diagramme de système

S : diagrama de sistema

A *system diagram* is the *SDL/GR* representation of a *system.* [2.4.2]

task

F : tâche

S : tarea

A *task* is an action within a *transition* containing either a sequence of *assignment statements* or *informal text.* The interpretation of a *task* depends on and may act on information held by the *system.* [2.7.1]

task area

F : zone de tâche

S : área de tarea

A *task area* is the SDL/GR representation of a *task.* [2.7.1]

term

F : terme

S : término

A *term* is syntactically equivalent to an *expression. Terms* are only used in *axioms* and are distinguished from *expressions* for reasons of clarity. [5.2.3, 5.3.3]

text extension symbol

F : symbole d'extension de texte

S : símbolo de ampliación de texto

A *text extension symbol* is a container of text which belongs to the *graphical symbol* to which the *text extension symbol* is attached. The text in the *text extension symbol* follows the text in the symbol to which it is attached. [2.2.7]

time

F : temps (time)

S : tiempo; time

Time is a *sort* defined in a predefined *partial type definition* for which the *values* are denoted as the *values* of *real.* The predefined *operators* using *time* and *duration* are + and −. [5.5.4.1, 5.6.12]

timer

F : temporisateur

S : temporizador

A *timer* is an object, owned by a *process instance,* that can be *active* or *inactive.* An *active timer* returns a *timer signal* to the owning *process instance* at a specified time. See also *set* and *reset.* [2.8, 5.5.4.5]

transition

F : transition

S : transición

A *transition* is an active sequence which occurs when a *process instance* changes from one *state* to another. [2.6.7.1]

transition area

F : zone de transition

S : àrea de transición

A *transition area* is the *SDL/GR* representation of a *transition.* [2.6.7.1]

transition string

F : chaîne de transition

S : cadena de transición

A *transition string* is a sequence of zero or more *actions.* [2.6.7.1]

transition string area

F : zone de chaîne de transition

F : área de cadena de transición

A *transition string area* is the *SDL/GR* representation of a *transition string.* [2.6.7.1]

type

F : type

S : tipo

A *type* is a set of properties for entities. Examples of classes of *types* in *SDL* include *blocks, channels, signal routes, signals,* and *systems.* [1.3.1]

type definition

F : définition de type

S ; definición de tipo

A *type definition* defines the properties of a *type* [1.3.1]

undefined

F : indéfini (undefined)

S : indefinido

Undefined is a "special" *value* of every *sort* which indicates that a *variable* of that *sort* has not yet been *assigned* a normal *value.* See *access.* [5.5.2.2]

valid input signal set

F : ensemble de signaux d'entrée valides

S : conjunto de señales de entrada válidas

The *valid input signal set* of a *process* is the list of all external *signals* handled by any *input* in the *process.* It consists of those *signals* in *signal routes* leading to the *process.* Compare with *complete valid input signal set.* [2.4.4, 2.5.2]

valid specification

F : spécification valide

S : especificación válida

A *valid specification* is a *specification* which follows the *concrete syntax* and static *well-formedness rules.* 1.3.3]

value

F : valeur

S : valor

A *value* of a *sort* is one of the values which are associated with a *variable* of that *sort,* and which can be used with an *operator* requiring a *value* of that *sort.* A *value* is the result of the interpretation of an *expression.* [2.3.3, 5.1.3]

variable

F : variable

S : variable

A *variable* is an entity owned by a *process instance* or *procedure instance* which can be associated with a *value* through an *assignment statement.* When *accessed,* a *variable* yields the last *value* which was assigned to it. [2.3.2]

variable definition

F : définition de variable

S : definición de variable

A *variable definition* is the indication that the *variable names* listed will be *visible* in the *process, procedure* or *service* containing the definition. [2.6.1.1]

view definition

F : définition de visibilité

S : definición de visión

A *view definition* defines a *variable identifier* in another *process* where it has the *revealed attribute.* This allows the viewing *process* to *access* the *value* of that *variable.* [2.6.1.2]

view expression

F : expression de vue

S : expresión de visión

A *view expression* is used within an *expression* to yield the current *value* of a *viewed variable.* [5.5.4.4]

visibility

F : visibilité

S : visibilidad

The *visibility* of an *identifier* is the *scope units* in which it may be used. No two definitions in the same *scope unit* and belonging to the same *entity class* may have the same *name.* [2.2.2]

well-formedness rules

F : règles de bonne formation

S : reglas de formación correcta

Well-formedness rules are constraints on a *concrete syntax* enforcing static conditions not directly expressed by the syntax rules. [1.4.1, 1.4.2]

ANNEX B
(To Recommendation Z.100)

Abstract syntax summary

Identifier :: *Qualifier Name*

Qualifier = *Path-item* +

Path-item = *System-qualifier* |
Block-qualifier |
Block-Substructure-qualifier |
Signal-qualifier |
Process-qualifier |
Procedure-qualifier |
Sort-qualifier

System-qualifier :: *System-name*

Block-qualifier :: *Block-name*

Block-Substructure-qualifier :: *Block-substructure-name*

Process-qualifier :: *Process-name*

Procedure-qualifier :: *Procedure-name*

Signal-qualifier :: *Signal-name*

Sort-qualifier :: *Sort-name*

Name :: *Token*

Informal-text :: ...

System-definition :: *System-name*
Block-definition-**set**
Channel-definition-**set**
Signal-definition-**set**
Data-type-definition
Syn-type-definition-**set**

System-name = *Name*

Block-definition :: *Block-name*
Process-definition-**set**
Signal-definition-**set**
Channel-to-route-connection-**set**
Signal-route-definition-**set**
Data-type-definition
Syn-type-definition-**set**
[*Block-substructure-definition*]

Block-name = *Name*

Process-definition :: *Process-name*
Number-of-instances
Process-formal-parameter *
Procedure-definition-**set**
Signal-definition-**set**
Data-type-definition
Syn-type-definition-**set**
Variable-definition-**set**
View-definition-**set**
Timer-definition-**set**
Process-graph

Number-of-instances :: *Intg Intg*

Process-name = *Name*

Process-graph :: *Process-start-node*
State-node-**set**

Process-formal-parameter :: *Variable-name*
Sort-reference-identifier

Procedure-definition :: *Procedure-name*
*Procedure-formal-parameter**
Procedure-definition-**set**
Data-type-definition
Syn-type-definition-**set**
Variable-definition-**set**
Procedure-graph

Procedure-name = *Name*

Procedure-formal-parameter = *In-parameter* |
Inout-parameter

In-parameter :: *Variable-name*
Sort-reference-identifier

Inout-parameter :: *Variable-name*
Sort-reference-identifier

Procedure-graph :: *Procedure-start-node*
State-node-**set**

Procedure-start-node :: *Transition*

Channel-definition :: *Channel-name*
Channel-path
[*Channel-path*]

Channel-path :: *Originating-block*

		Destination-block *Signal-identifier*-**set**
Originating-block	=	*Block-identifier* | ENVIRONMENT
Destination-block	=	*Block-identifier* | ENVIRONMENT
Block-identifier	=	*Identifier*
Signal-identifier	=	*Identifier*
Channel-name	=	*Name*
Signal-route-definition	::	*Signal-route-name* *Signal-route-path* [*Signal-route-path*]
Signal-route-path	::	*Originating-process* *Destination-process* *Signal-identifier*-**set**
Originating-process	=	*Process-identifier* | ENVIRONMENT
Destination-process	=	*Process-identifier* | ENVIRONMENT
Signal-route-name	=	*Name*
Channel-to-route-connection	::	*Channel-identifier* *Signal-route-identifier*-**set**
Signal-route-identifier	=	*Identifier*
Signal-definition	::	*Signal-name* *Sort-reference-identifier** [*Signal-refinement*]
Signal-name	=	*Name*
Variable-definition	::	*Variable-name* *Sort-reference-identifier* [REVEALED]
Variable-name	=	*Name*
View-definition	::	*Variable-identifier* *Sort-reference-identifier*
Process-start-node	::	*Transition*

State-node :: *State-name*
Save-signalset
Input-node-**set**

State-name = *Name*

Input-node :: *Signal-identifier*
[*Variable-identifier*]*
Transition

Variable-identifier = *Identifier*

Save-signalset :: *Signal-identifier*-**set**

Transition :: *Graph-node* *
(*Terminator* | *Decision-node*)

Graph-node :: *Task-node* |
Output -node |
Create-Request-node |
Call-node |
Set-node |
Reset-node

Terminator :: *Nextstate-node* |
Stop-node |
Return-node

Nextstate-node :: *State-name*

Return-node :: ()

Stop-node :: ()

Task-node :: *Assignment-statement* |
Informal -text

Create-request-node :: *Process-identifier*
[*Expression*]*

Process-identifier = *Identifier*

Call-node :: *Procedure-identifier*
[*Expression*] *

Procedure-identifier = *Identifier*

Decision-node :: *Decision-question*
Decision-answer-**set**
[*Else-answer*]

Decision-question = *Expression* |

Informal-text

Decision-answer :: *(Range-condition* | *Informal-text) Transition*

Else-answer :: *Transition*

Output-node :: *Signal-identifier*
[*Expression*]*
[*Signal-destination*]
Direct-via

Signal-destination = *Expression*

Direct-via = *Signal-route-identifier*-**set**

Timer-definition :: *Timer-name Sort-reference-identifier**

Timer-name = *Name*

Set-node :: *Time-expression*
Timer-identifier
*Expression**

Reset-node :: *Timer-identifier*
*Expression**

Timer-identifier = *Identifier*

Time-expression = *Expression*

Block-substructure-definition :: *Block-substructure-name*
Sub-block-definition-**set**
Channel-connection-**set**
Channel-definition-**set**
Signal-definition-**set**
Data-type-definition
Syn-type-definition-**set**

Block-substructure-name = *Name*

Sub-block-definition = *Block-definition*

Channel-connection :: *Channel-identifier*
Sub-channel-identifier-**set**

Sub-channel-identifier = *Channel-identifier*

Channel-identifier = *Identifier*

Signal-refinement :: *Subsignal-definition*-**set**

Subsignal-definition :: [REVERSE] *Signal-definition*

Data-type-definition ::
Type-name
Type-union
Sorts
Signature–**set**
Equations

Type-union = *Type-identifier*–**set**

Type-identifier = *Identifier*

Sorts = *Sort-name*-**set**

Type-name = *Name*

Sort-name = *Name*

Equations = *Equation*–**set**

Signature = *Literal-signature* |
Operator-signature

Literal-signature ::
Literal-operator-name
Result

Operator-signature ::
Operator-name
Argument-list
Result

Argument-list = *Sort-reference-identifier* $^{+}$

Result = *Sort-reference-identifier*

Sort-reference-identifier = *Sort-identifier* |
Syntype-identifier

Literal-operator-name = *Name*

Operator-name = *Name*

Sort-identifier = *Identifier*

Equation = *Unquantified-equation* |
Quantified-equations |
Conditional-equation |
Informal-text

Unquantified-equation ::
Term
Term

Quantified-equations	::	*Value-name*–**set** *Sort-identifier* *Equations*
Value-name	=	*Name*
Term	=	*Ground-term* | *Composite-term* | *Error-term*
Composite-term	::	*Value-identifier* | *Operator-identifier Term*$^{+}$ | *Conditional-composite-term*
Value-identifier	=	*Identifier*
Operator-identifier	=	*Identifier*
Ground-term	::	*Literal-operator-identifier* | *Operator-identifier Ground-term*$^{+}$ | *Conditional-ground-term*
Literal-operator-identifier	=	*Identifier*
Conditional-equation	::	*Restriction*–**set** *Restricted-equation*
Restriction	=	*Unquantified-equation*
Restricted-equation	=	*Unquantified-equation*
Conditional-composite-term	=	*Conditional-term*
Conditional-ground-term	=	*Conditional-term*
Conditional-term	::	*Condition* *Consequence* *Alternative*
Condition	=	*Term*
Consequence	=	*Term*
Alternative	=	*Term*
Error-term	::	()
Syntype-identifier	=	*Identifier*
Syn-type-definition	::	*Syntype-name* *Parent-sort-identifier* *Range-condition*

Syntype-name	=	*Name*
Parent-sort-identfier	=	*Sort-identifier*
Range-condition	::	*Or-operator-identifier* *Condition-item*–**set**
Condition-item	=	*Open-range* | *Closed-range*
Open-range	::	*Operator-identifier* *Ground-expression*
Closed-range	::	*And-operator-identifier* *Open-range* *Open-range*
Or-operator-identifier	=	*Identifier*
And-operator-identifier	=	*Identifier*
Expression	=	*Ground-expression* | *Active-expression*
Ground-expression	::	*Ground-term*
Variable-access	=	*Variable-identifier*
Active-expression	=	*Variable-access* | *Conditional-expression* | *Operator-application* | *Imperative-operator*
Imperative-operator	=	*Now-expression* | *Pid-expression* | *View-expression* | *Timer-active-expression*
Now-expression	::	()
Pid-expression	=	*Self-expression* | *Parent-expression* | *Offspring-expression* | *Sender-expression*
Self-expression	::	()
Parent-expression	::	()
Offspring-expression	::	()
Sender-expression	::	()

View-expression	::	*Variable-identifier* *Expression*
Timer-active-expression	::	*Timer-identifier* *Expression**
Conditional-expression	::	*Boolean-expression* *Consequence-expression* *Alternative-expression*
Boolean-expression	=	*Expression*
Consequence-expression	=	*Expression*
Alternative-expression	=	*Expression*
Operator-application	::	*Operator-identifier* *Expression*$^{+}$
Assignment-statement	::	*Variable-identifier* *Expression*

ANNEX C1
(To Recommendation Z.100)

Concrete graphical syntax summary

C1.1 Introduction

C1.1.1 *Metalanguage*

For the graphical grammar the metalanguage described in SDL Rec § 1.5.2 is extended with the following metasymbols:

a) **contains**
b) **is associated with**
c) **is followed by**
d) **is connected to**
e) **set**

The **set** metasymbol is a postfix operator operating on the immediately preceding syntactic elements within curly brackets, and indicating an (unordered) set of items. Such items may be any syntactic element, in which case it must be applied before the **set** metasymbol. Example:

{{<system text area>}* {<macro diagram>}* <block interaction area>} **set**

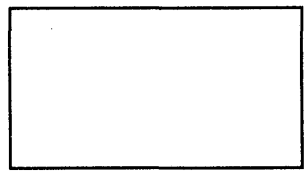
is a set of zero or more <system text area>s, zero or more <macro diagram>s and one <block interaction area>.

All the other metasymbols are infix operators, having a graphical non-terminal symbol as the left-hand argument. The right-hand argument is either a group of syntactic elements within curly brackets or a single syntactic element. If the right-hand side of a production rule has a graphical non-terminal symbol as the first element and contains one or more of these infix operators, then the graphical non-terminal symbol is the left-hand argument of each of these infix operators. A graphical non-terminal symbol is a non-terminal having the word "symbol" immediately before the greater than sign >.

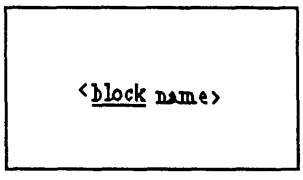
The metasymbol **contains** indicates that its right-hand argument should be placed within its left-hand argument and the attached <text extension symbol>, if any. Example:

<block reference> ::=
<block symbol> **contains** <block name>

<block symbol> ::=



means the following



The metasymbol **is associated with** indicates that its right-hand side argument is logically associated with its left-hand argument (as if it were "contained" in that argument, the unambiguous association is ensured by appropriate drawing rules).

The metasymbol **is followed by** means that its right-hand argument follows (both logically and in drawing) its left-hand argument.

The metasymbol **is connected to** means that its right-hand argument is connected (both logically and in drawing) to its left-hand argument.

C1.1.2 *General rules*

C1.1.2.1 *Partitioning of diagrams*

The following definition of diagram partitioning is not part of the *Concrete graphical grammar*, but the same meta-language is used.

<page>::=
<frame symbol> **contains**
<heading area> <page number area>
{<syntactical unit>}*

<heading area> ::=
<implicit text symbol> **contains** <heading>

<page number area> ::=
<implicit text symbol> **contains** [<page number> [(<number of pages>)]]

<page number> ::=
<literal name>

<number of pages> ::=
<natural literal name>

<page> is a starting non-terminal, therefore it is not referred to in any production rule. A diagram may be partitioned into a number of <page>s, in which case the <frame symbol> delimiting the diagram and the diagram <heading> are replaced by a <frame symbol> and a <heading> for each page.

The <implicit text symbol> is not shown, but implied, in order to have a clear separation

between <heading area> and <page number area>. <heading area> is placed at the upper left corner of the <frame symbol>. <page number area> is placed at the upper right corner of the <frame symbol>. <heading> and <syntactical unit> depends on the type of diagram.

C1.1.2.2 *Comment*

<comment area> ::=
<comment symbol> **contains** <text>
is connected to <dashed association symbol>

<comment symbol> ::=

<dashed association symbol> ::=

One end of the <dashed association symbol> must be connected to the middle of the vertical segment of the <comment symbol>.

A <comment symbol> is connected to any graphical symbol by means of a <dashed association symbol>. The <comment symbol> is considered as a closed symbol by completing (in imagination) the rectangle. It contains comment text related to the graphical symbol.

C1.1.2.3 *Text extension*

<text extension area> ::=
<text extension symbol> **contains** <text>
is connected to <solid association symbol>

<text extension symbol> ::=
<comment symbol>

<solid association symbol> ::=

A <text extension symbol> is connected to any graphical symbol by means of a <solid association symbol>. The <text extension symbol> is considered as a closed symbol by completing (in imagination) the rectangle.

One end of the <solid association symbol> must be connected to the middle of the vertical segment of the <text extension symbol>.

The text contained in the <text extension symbol> is a continuation of the text within the graphical symbol and is considered to be contained in that symbol.

C1.2 *System definition*

<concrete system definition> ::=
{<system definition> | <system diagram>} {<remote definition>}*

<remote definition> ::=
<definition>
| <diagram>

<diagram> ::=
<block diagram>
| <process diagram>
| <procedure diagram>
| <block substructure diagram>
| <channel substructure diagram>
| <service diagram>
| <macro diagram>

C1.3 *System diagram*

<system diagram> ::=
<frame symbol> **contains**
{<system heading>
{ {<system text area>}*
{<macro diagram>}*
<block interaction area> }**set** }

<frame symbol> ::=

<system heading> ::=
SYSTEM <system name>

<system text area> ::=
<text symbol> **contains**
{<signal definition>
| <signal list definition>
| <data definition>
| <macro definition>
| <select definition>}*

<text symbol> ::=

<block interaction area> ::=
{<block area> | <channel definition area>}+

<block area> ::=
<graphical block reference>
| <block diagram>

<graphical block reference> ::=
<block symbol> **contains** <block name>

<block symbol> ::=

<channel definition area> ::=
<channel symbol>
is associated with {<channel name>
{ [{<channel identifier> | <block identifier>}]
<signal list area>
[<signal list area>] }**set** }
is connected to { <block area>
{<block area> | <frame symbol>}
[<channel substructure association area>] }**set**

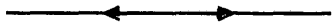
The <channel identifier> identifies an external channel connected to the <block substructure diagram> delimited by the <frame symbol>. The <block identifier> identifies an external block being a channel endpoint for the <channel substructure diagram> delimited by the <frame symbol>.

<channel symbol> ::=
<channel symbol 1>
| <channel symbol 2>
| <channel symbol 3>

<channel symbol 1> ::=

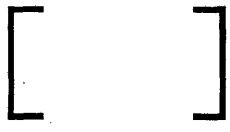
<channel symbol 2> ::=

<channel symbol 3> ::=



<signal list area> ::=
<signal list symbol> **contains** <signal list>

<signal list symbol> ::=



C1.4 *Block diagram*

<block diagram> ::=
<frame symbol>
contains {<block heading>
{ {<block text area>}*
{<macro diagram>}*
[<process interaction area>]
[<block substructure area>] }**set** }
is associated with {<channel identifier>}*

The <channel identifier> identifies a channel connected to a signal route in the <block diagram>. It is placed outside the <frame symbol> close to the endpoint of the signal route at the <frame symbol>. If the <block diagram> does not contain a <process interaction area>, then it must contain a <block substructure area>.

<block heading> ::=
BLOCK {<block name> | <block identifier> }

<block text area> ::=
<system text area>

<process interaction area> ::=
{ <process area>
| <create line area>
| <signal route definition area> }+

<process area> ::=
<graphical process reference>
| <process diagram>

<graphical process reference> ::=
<process symbol> **contains** {<process name> [<number of instances>]}

<process symbol> ::=

<create line area> ::=
<create line symbol>
is connected to {<process area> <process area>}

<create line symbol> ::=

<signal route definition area> ::=
<signal route symbol>
is associated with {<signal route name>
{ [<channel identifier>]
<signal list area>
[<signal list area>] }**set** }
is connected to
{<process area> {<process area> | <frame symbol>} }**set**

When the <signal route symbol> is connected to the <frame symbol>, then the <channel identifier> identifies a channel to which the signal route is connected.

<signal route symbol> ::=
<signal route symbol 1>
| <signal route symbol 2>

<signal route symbol 1> ::=

<signal route symbol 2> ::=

C1.5 *Process diagram*

<process diagram> ::=
<frame symbol>
contains {<process heading>
{ {<process text area>}*
{<procedure area>}*
{<macro diagram> }*
{<process graph area> | <service interaction area> } }**set** }
[**is associated with** {<signal route identifier>}+]

The <signal route identifier> identifies an external signal route connected to a signal route in the <process diagram>. It is placed outside the <frame symbol> close to the endpoint of the signal route at the <frame symbol>.

<process heading> ::=
PROCESS {<process name> | <process identifier>}
[<number of instances> <end>] [<formal parameters>]

<process text area> ::=
<text symbol> **contains** [<valid input signal set>]
{<signal definition>
| <signal list definition>
| <variable definition>
| <view definition>
| <import definition>
| <data definition>
| <macro definition>
| <timer definition>
| <select definition> }*

<graphical procedure reference> ::=
<procedure symbol> **contains** <procedure name>

<procedure symbol> ::=

<process graph area> ::=
<start area> { <state area> |<in-connector area> }*

<start area> ::=
<start symbol> **is followed by** <transition area>

<start symbol> ::=

<transition area> ::=
[<transition string area>] **is followed by**
{<state area>
| <nextstate area>
| <decision area>
| <stop symbol>
| <merge area>
| <out-connector area>
| <return symbol>
| <transition option area> }

<merge area> ::=
<merge symbol> **is connected to** <flow line symbol>

<merge symbol> ::=
<flow line symbol>

<flow line symbol> ::=

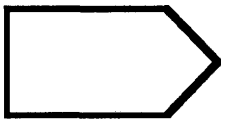
<transition string area> ::=
{<task area>
| <output area>
| <priority output area>
| <set area>
| <reset area>
| <export area>
| <create request area>
| <procedure call area> }
[**is followed by** <transition string area>]

<task area> ::=
<task symbol> **contains** <task body>

<task symbol> ::=

<output area> ::=
<output symbol> **contains** <output body>

<output symbol> ::=



<create request area> ::=
<create request symbol> **contains** <create body>

<create request symbol> ::=

<procedure call area> ::=
<procedure call symbol> **contains** <procedure call body>

<procedure call symbol> ::=

<state area> ::=
<state symbol> **contains** <state list> **is associated with**
{<input association area>
| <priority input association area>
| <continuous signal association area>
| <save association area> }*

<state symbol> ::=

<input association area> ::=
<solid association symbol> **is connected to** <input area>

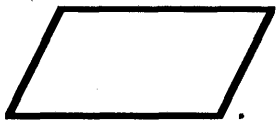
<input area> ::=
<input symbol> **contains** <input list>
is followed by { [<enabling condition area>] <transition area>}

<input symbol> ::=

<save association area> ::=
<solid association symbol> **is connected to** <save area>

<save area> ::=
<save symbol> **contains** <save list>

<save symbol> ::=



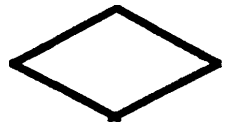
<in-connector area> ::=
<in-connector symbol> **contains** <connector name>
is followed by <transition area>

<in-connector symbol> ::=

<nextstate area> ::=
<state symbol> **contains** <nextstate body>

<decision area> ::=
<decision symbol> **contains** <question>
is followed by
{ {<graphical answer part> <graphical else part>} **set**
| {<graphical answer part> {<graphical answer part>}+ [<graphical else part>] } set }

<decision symbol> ::=



<graphical answer> ::=
<answer> | (<answer>)

<graphical answer part> ::=
<flow line symbol> **is associated with** <graphical answer>
is followed by <transition area>

<graphical else part> ::=
<flow line symbol> **is associated with** ELSE
is followed by <transition area>

<set area>::=
<task symbol> **contains** <set>

<reset area>
<task symbol> **contains** <reset>

<stop symbol> ::=

<out-connector area> ::=
<out-connector symbol> **contains** <connector name>

<out-connector symbol> ::=
<in-connector symbol>

C1.6 · *Procedure diagram*

<procedure diagram> ::=
<frame symbol> **contains** {<procedure heading>
{ {<procedure text area>}*
{ <procedure area>}*
{<macro diagram> }*
<procedure graph area> }**set** }

<procedure heading> ::=
PROCEDURE {<procedure name> | <procedure identifier> }
[<procedure formal parameters>]

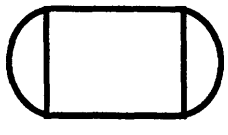
<procedure area> ::=
<graphical procedure reference>
| <procedure diagram>

<procedure text area> ::=
<text symbol> **contains**
{<variable definition>
| <data definition>
| <macro definition>
| <select definition> }*

<procedure graph area> ::=
<procedure start area> {<state area> | <in-connector area> }*

<procedure start area> ::=
<procedure start symbol> **is followed by** <transition area>

<procedure start symbol> ::=



<return symbol> ::=

C1.7 *Block substructure*

<block substructure area> ::=
<graphical block substructure reference>
| <block substructure diagram>

<graphical block substructure reference> ::=
<block substructure symbol> **contains** <block substructure name>

<block substructure symbol> ::=
<block symbol>

<block substructure diagram> ::=
<frame symbol>
contains {<block substructure heading>
{ {<block substructure text area>}*
{<macro diagram>}*
<block interaction area> }**set** }
is associated with {<channel identifier>}*

The <channel identifier> identifies a channel connected to a subchannel in the <block substructure diagram>. It is placed outside the <frame symbol> close to the endpoint of the subchannel at the <frame symbol>.

<block substructure heading> ::=
SUBSTRUCTURE {<block substructure name> | <block substructure identifier>}

<block substructure text area> ::=
<system text area>

C1.8 *Channel substructure*

<channel substructure association area> ::=
<dashed association symbol>
is connected to <channel substructure area>

<channel substructure area> ::=
<graphical channel substructure reference>
| <channel substructure diagram>

<graphical channel substructure reference> ::=
<channel substructure symbol> **contains** <channel substructure name>

<channel substructure symbol> ::=
<block symbol>

<channel substructure diagram> ::=
<frame symbol>
contains {<channel substructure heading>
{ {<channel substructure text area>}*
{<macro diagram>}*
<block interaction area> }**set** }
is associated with {<block identifier> | ENV}+

The <block identifier> or ENV identifies an endpoint of the partitioned channel. The <block identifier> is placed outside the <frame symbol> close to the endpoint of the associated subchannel at the <frame symbol>.

<channel substructure heading> ::=
SUBSTRUCTURE { < channel substructure name>
| < channel substructure identifier>}

<channel substructure text area> ::=
<system text area>

C1.9 *Macro*

9.1 *Macro diagram*

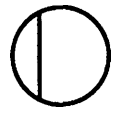
<macro diagram> ::=
<frame symbol> **contains** <macro heading> <macro body area>

<macro heading> ::=
MACRODEFINITION <macro name> [<macro formal parameters>]

<macro body area> ::=
{ {<any area> }*
<any area> [**is connected to** <macro body port1>] }**set**
|{ <any area> **is connected to** <macro body port2>
<any area> **is connected to** <macro body port2>
{ <any area> [**is connected to** <macro body port2>]}*}**set**

<macro inlet symbol>::=

<macro outlet symbol>::=



<macro body port1> ::=
<outlet symbol> [**is associated with** <macro label>]
is connected to {<frame symbol>
| <macro inlet symbol>
| <macro outlet symbol>}

<macro body port2> ::=
<outlet symbol> **is associated with** <macro label>
is connected to {<frame symbol>
| <macro inlet symbol>
| <macro outlet symbol>}

<macro label> ::=
<name>

<outlet symbol> ::=
<dummy outlet symbol>
| <flow line symbol>
| <channel symbol>
| <signal route symbol>
| <solid association symbol>
| <dashed association symbol>
| <create line symbol>

<dummy outlet symbol> ::=
<solid association symbol>

<any area> ::=
<system text area>
| <block interaction area>
| <signal list area>
| <block area>
| <block text area>
| <process interaction area>
| <graphical procedure reference>
| <procedure area>
| <process text area>
| <process graph area>
| <merge area>
| <transition string area>
| <state area>
| <input area>
| <save area>
| <text extension area>
| <channel substructure association area>
| <channel substructure area>
| <block substructure area>
| <priority input area>
| <continuous signal area>
| <in-connector area>
| <nextstate area>
| <process area>
| <channel definition area>
| <create line area>
| <signal route definition area>
| <graphical process reference>
| <process diagram>
| <start area>
| <output area>
| <set area>
| <reset area>
| <export area>
| <priority output area>
| <task area>
| <create request area>
| <procedure call area>
| <decision area>

| <out-connector area>
| <procedure text area>
| <procedure graph area>
| <procedure start area>
| <block substructure text area>
| <block interaction area>
| <service area>
| <service signal route definition area>
| <service text area>
| <service graph area>
| <service start area>
| <comment area>
| <macro call area>

C1.9.2 *Macro call*

<macro call area> ::=
<macro call symbol> **contains** {<macro name> [<macro call body>]}
[**is connected to**
{<macro call port1> | <macro call port2> {<macro call port2>}+}]

<macro call symbol> ::=

<macro call port1> ::=
<inlet symbol> [**is associated with** <macro label>]
is connected to <any area>

<macro call port2> ::=
<inlet symbol> **is associated with** <macro label>
is connected to <any area>

<inlet symbol> ::=
<dummy inlet symbol>
| <flow line symbol>
| <channel symbol>
| <signal route symbol>
| <solid association symbol>
| <dashed association symbol>
| <create line symbol>

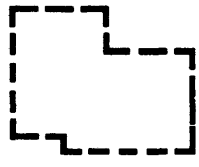
<dummy inlet symbol> ::=
<solid association symbol>

C1.10 *Generic systems*

C1.10.1 *Optional definition*

<option area> ::=
<option symbol> **contains**
{ SELECT IF (<boolean simple expression>)
{<block area>
| <channel substructure area>
| <system text area>
| <block text area>
| <process text area>
| <procedure text area>
| <block substructure text area>
| <channel substructure text area>
| <service text area>
| <macro diagram>
| <option area>
| <process area>
| <signal route definition area>
| <create line area>
| <procedure area>
| <service area>
| <service signal route definition area> }+ }

The <option symbol> is a dashed polygon having solid corners, for example:



C1.10.1 *Optional transition*

<transition option area> ::=
<transition option symbol> **contains** {<alternative question>}
is followed by {<option outlet1> {<option outlet1> | <option outlet2> }
{ <option outlet1> }* }**set**

<transition option symbol> ::=

<option outlet1> ::=
<flow line symbol> **is associated with** <graphical answer>
is followed by {<transition area> | <merge area>}

<option outlet2> ::=
<flow line symbol> **is associated with** ELSE
is followed by {<transition area> | <merge area>}

C1.11 *Service*

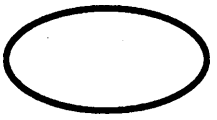
C1.11.1 *Service decomposition*

<service interaction area> ::=
{ <service area> | <service signal route definition area> }+

<service area> ::=
<graphical service reference>
| <service diagram>

<graphical service reference> ::=
<service symbol> **contains** <service name>

<service symbol> ::=



<service signal route definition area> ::=
<signal route symbol>
is associated with {<service signal route name> [<signal route identifier>]
<signal list area> [<signal list area>]}**set**
is connected to {<service area>
{<service area> | <frame symbol>} }**set**

When the <signal route symbol> is connected to the <frame symbol>, then the <signal route identifier> identifies an external signal route to which the signal route is connected.

C1.11.2 *Service diagram*

<service diagram> ::=
<frame symbol> **contains**
{<service heading>
{ {<service text area>}*
{<procedure area>}*
{<macro diagram>}*
<service graph area> }**set** }

<service heading> ::=
SERVICE { <service name> | <service identifier>}

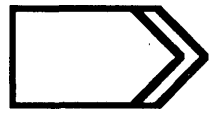
<service text area> ::=
<text symbol> **contains**
{ <variable definition>
| <view definition>
| <import definition>
| <data definition>
| <macro definition>
| <timer definition>
| <select definition> }*

<service graph area> :::=
<process graph area>

<priority input association area> ::=
<solid association symbol> **is connected to** <priority input area>

<priority input area> ::=
<priority input symbol> **contains** <priority input list>

<priority input symbol> ::=



<priority output area> ::=
<priority output symbol> **contains** <priority output body>

<priority output symbol> ::=

C1.12 *Continuous signal*

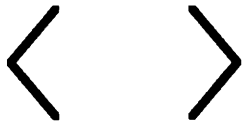
<continuous signal association area> ::=
<solid association symbol> **is connected to** <continuous signal area>

<continuous signal area> ::=
<enabling condition symbol>
contains {<boolean expression> [<end> PRIORITY <integer literal name>]}
is followed by <transition area>

C1.13 *Enabling condition*

<enabling condition area> ::=
<enbling condition symbol> **contains** <boolean expression>

<enabling condition symbol> ::=



C1.14 *Export*

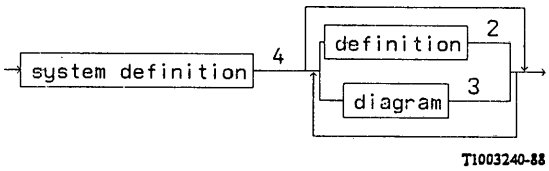
<export area> ::=
<task symbol> **contains** <export>

ANNEX C2

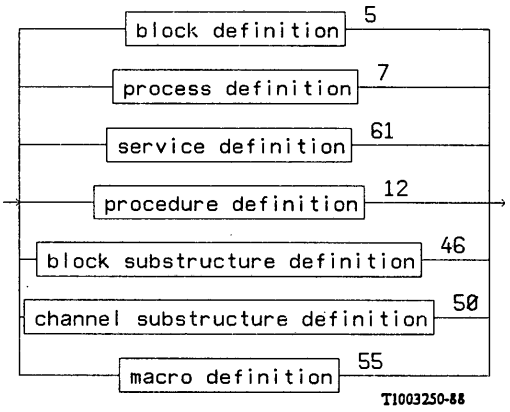
(to Recommendation Z.100)

SDL PR syntax summary

1 *system*



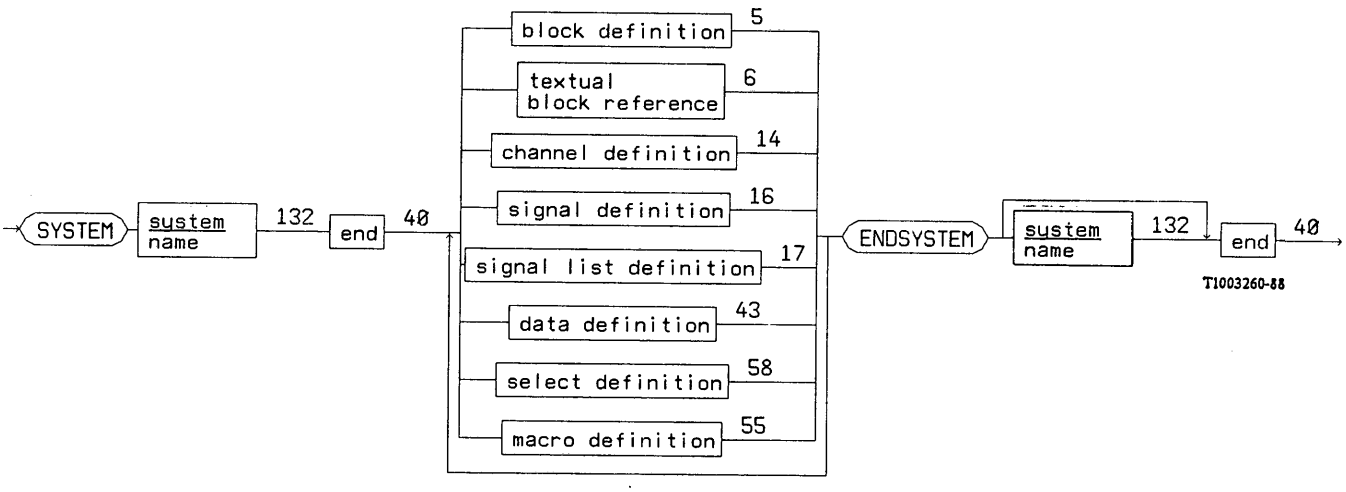
2 *definition*



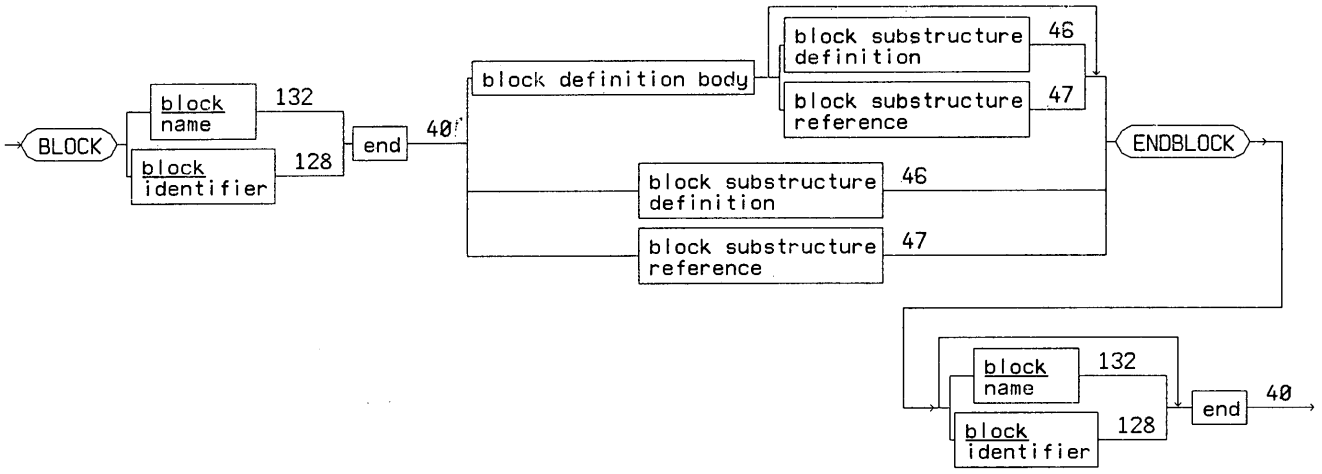
3 *diagram*

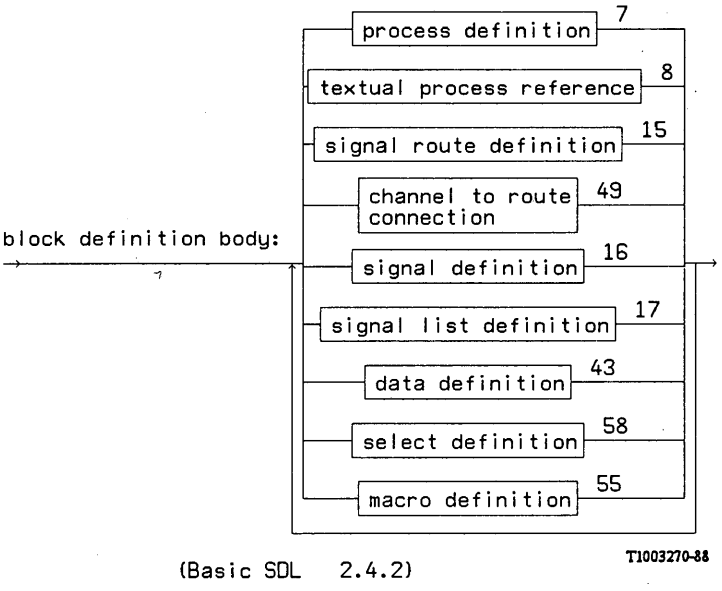
Diagram is defined in GR syntax summary.

4 *system definition* (Basic SDL 2.4.1)



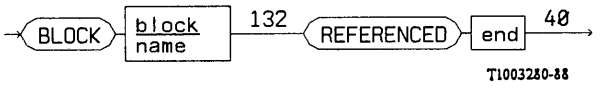
5 *block definition* (Basic SDL 2.4.2)

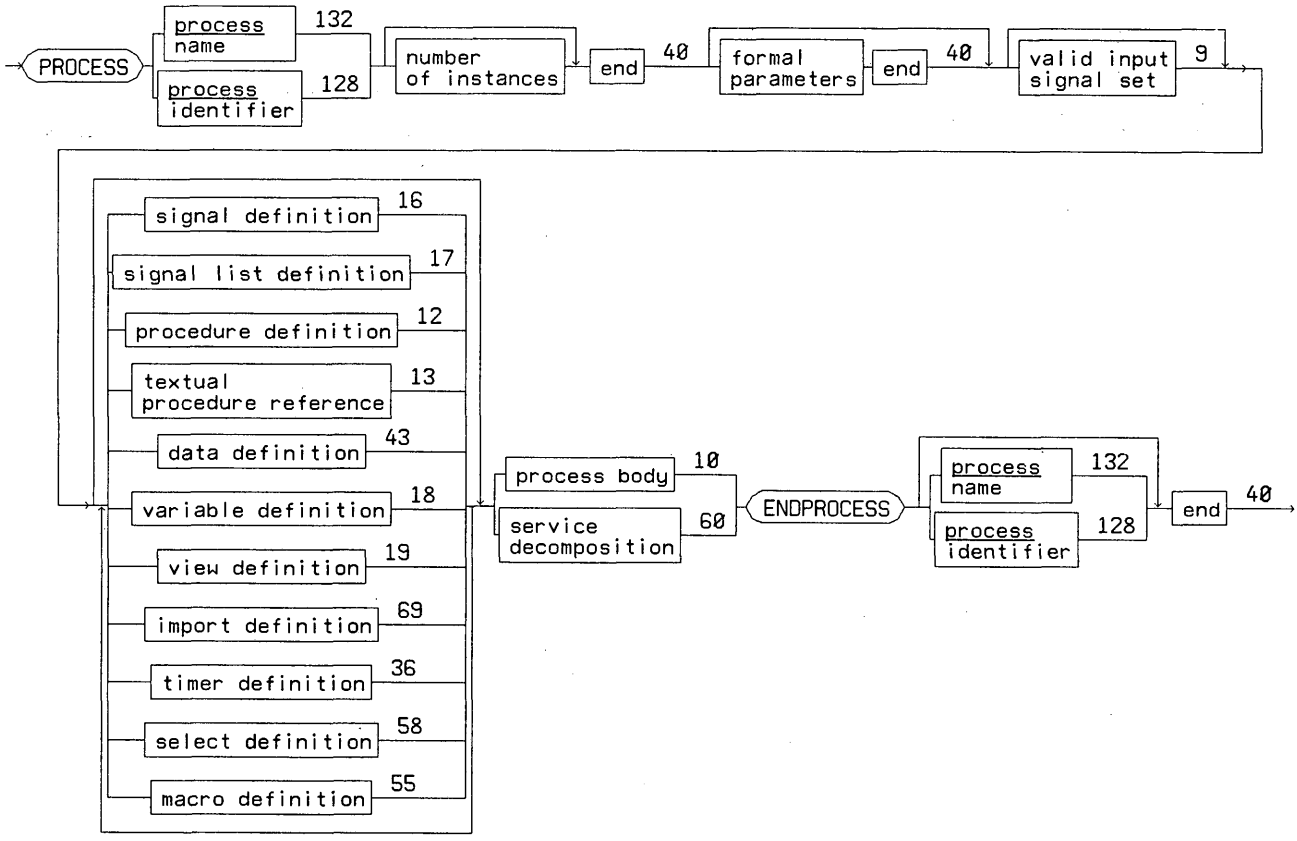


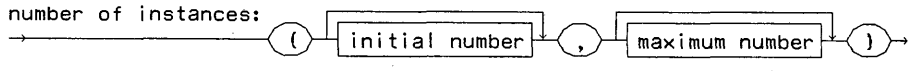


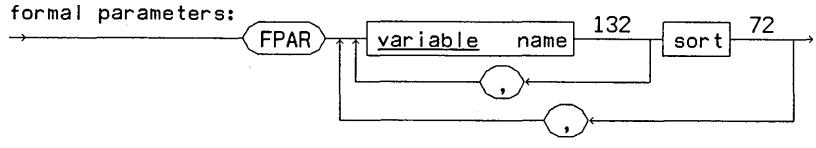
(Basic SDL 2.4.2)

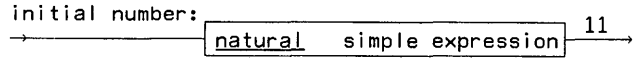
6 *textual block reference* (Basic SDL 2.4.2)

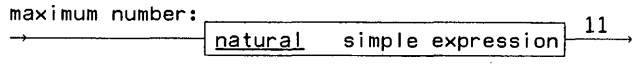



PROCESS
process name
132
process identifier
128
number of instances
end
40
formal parameters
end
40
valid input signal set
9
signal definition
16
signal list definition
17
procedure definition
12
textual procedure reference
13
data definition
43
variable definition
18
view definition
19
import definition
69
timer definition
36
select definition
58
macro definition
55
process body
10
service decomposition
60
ENDPROCESS
process name
132
process identifier
128
end
40


number of instances:
(
initial number
,
maximum number
)

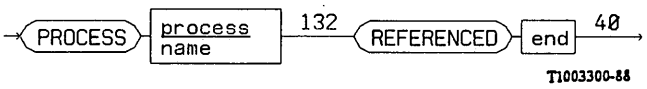

formal parameters:
FPAR
variable name
132
sort
72
,
,


initial number:
natural simple expression
11

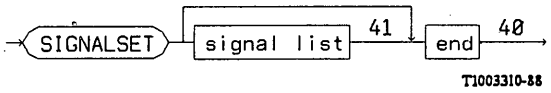

maximum number:
natural simple expression
11

T1003290-88

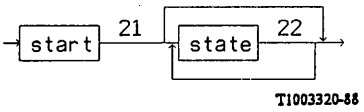
8 *textual process reference* (Basic SDL 2.4.3)



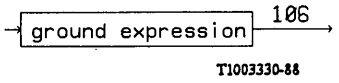
9 *valid input signal set*



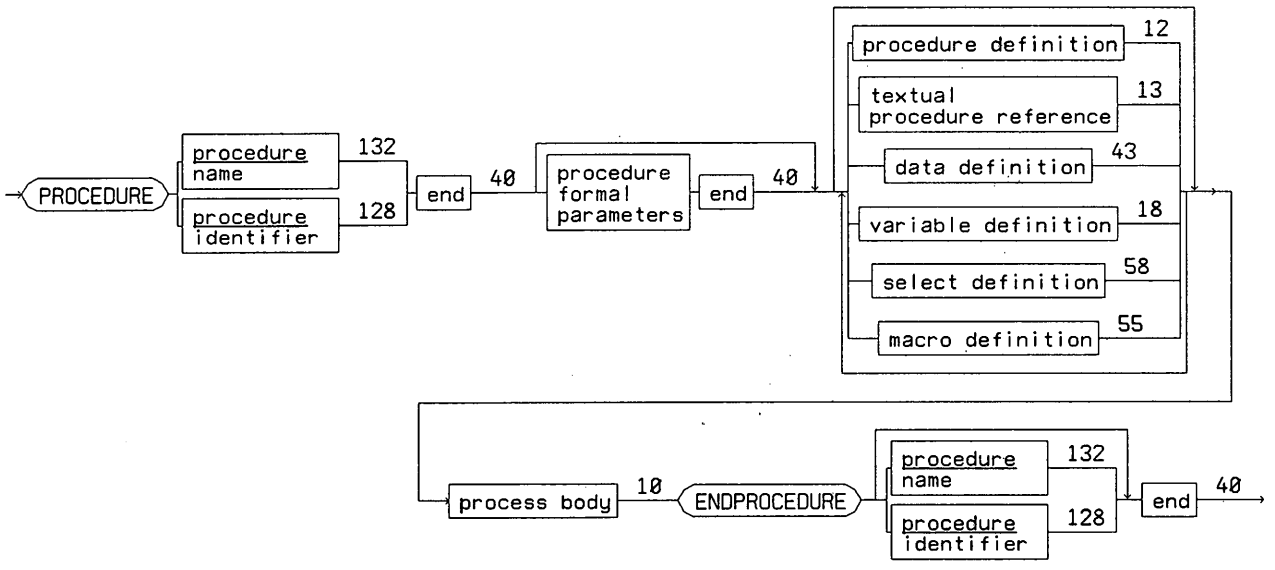
10 *process body* (Basic SDL 2.4.3)

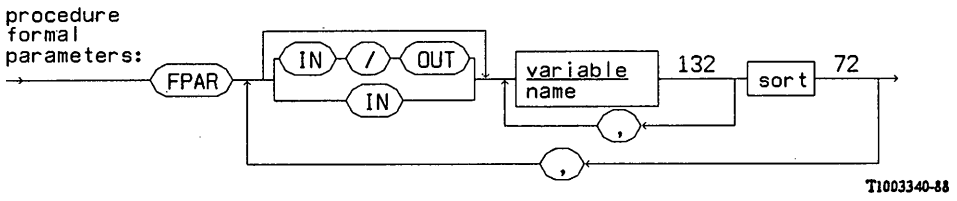


11 *simple expression*

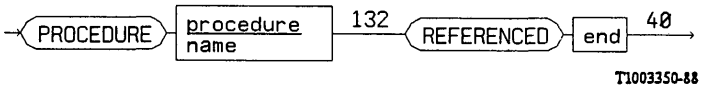


12 *procedure definition* (Basic SDL 2.3.4)

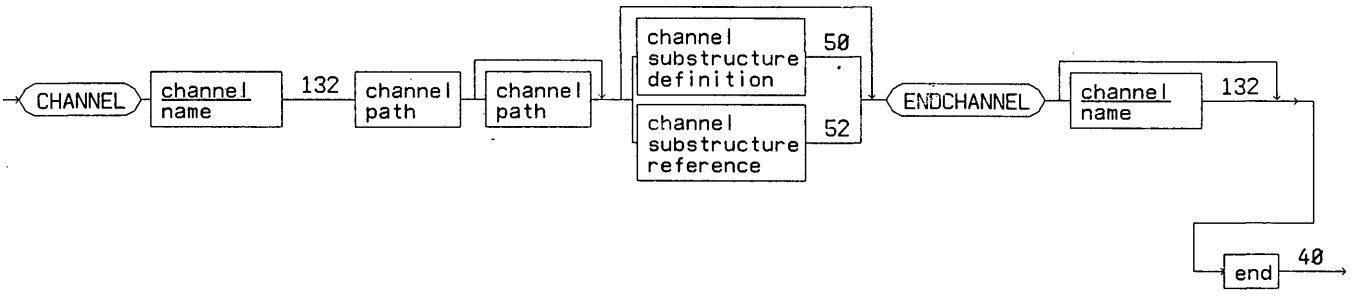


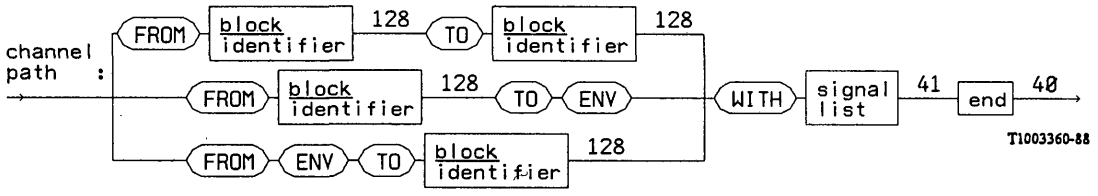


13 *textual procedure reference* (Basic SDL 2.3.4)

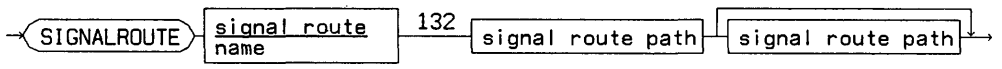


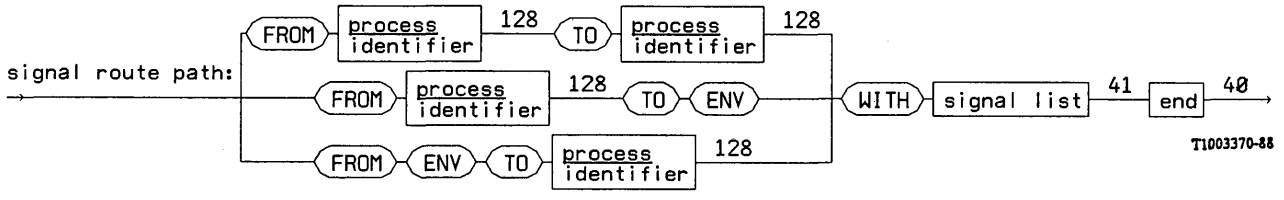
14 *channel definition* (Basic SDL 2.5.1)



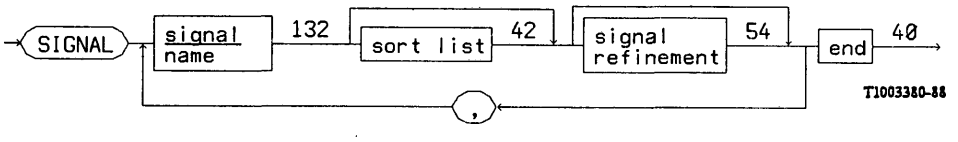


15 *signal route definition* (Basic SDL 2.5.2)

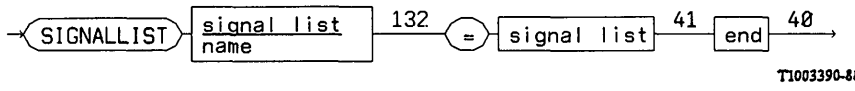




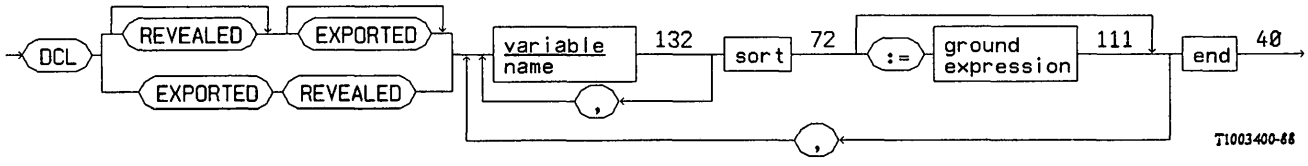
16 *signal definition* (Basic SDL 2.5.4)



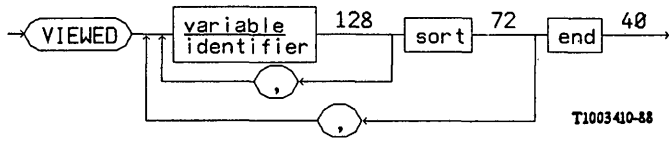
17 *signal list definition* (Basic SDL 2.5.5)



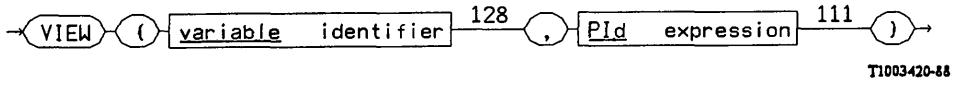
18 *variable definition* (Basic SDL 2.6.1.1)



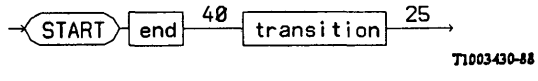
19 *view definition* (Basic SDL 2.6.1.2)



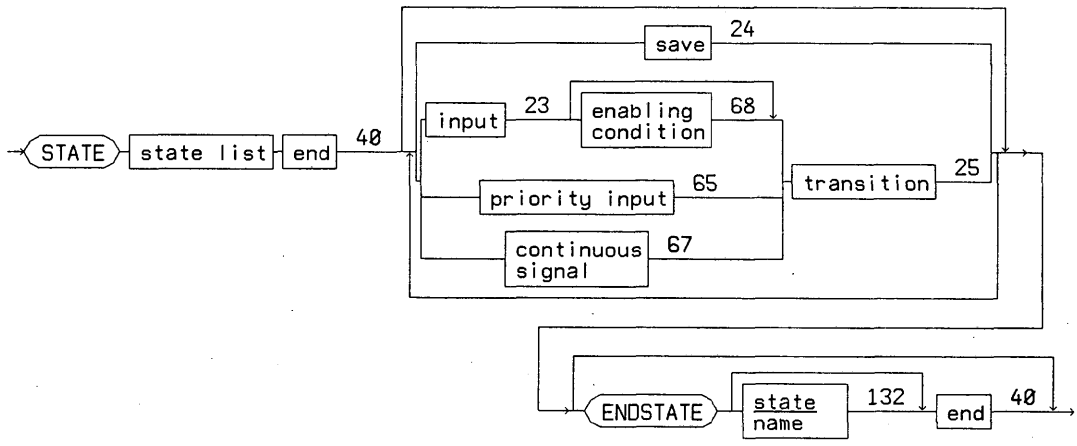
20 *view expression* (Data 5.5.4.4)



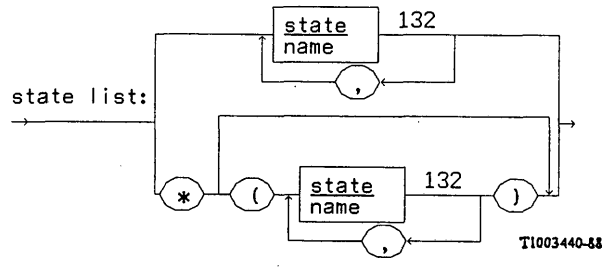
21 *start* (Basic SDL 2.6.2)



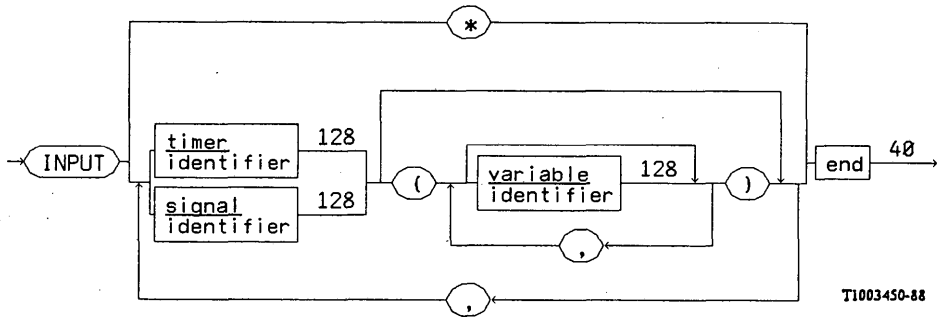
22 *state* (Basic SDL 2.6.3)



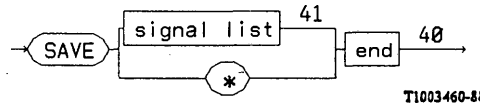
(Additional concepts 4.4)



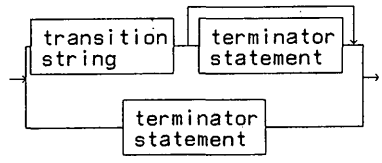
23 *input* (Basic SDL 2.6.4)

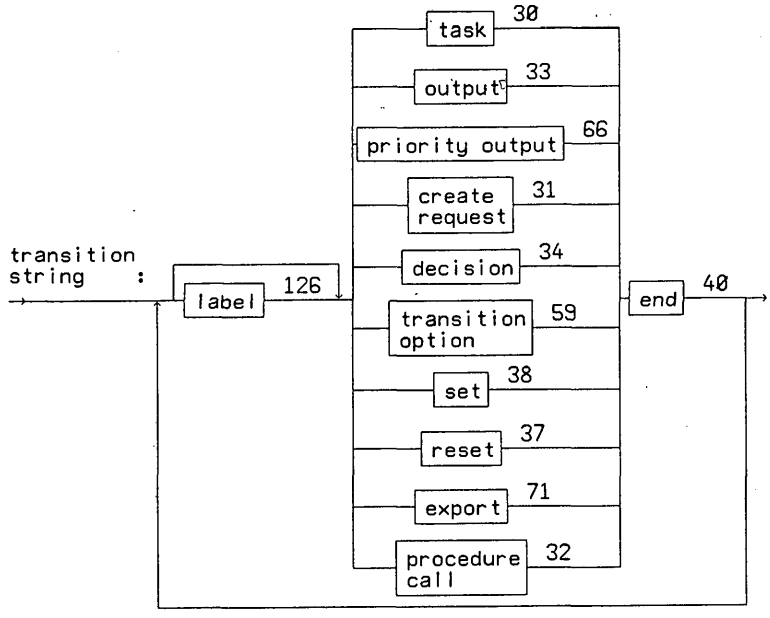


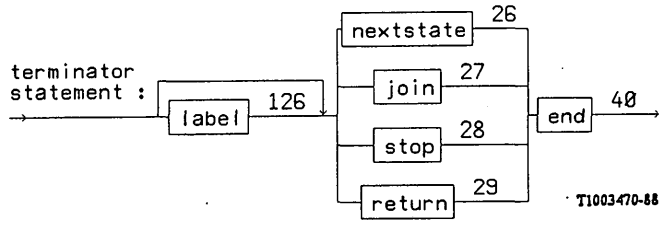
24 *save* (Basic SDL 2.6.5)



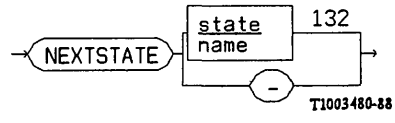
25 *transition* (Basic SDL 2.6.7)



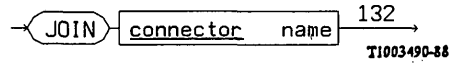




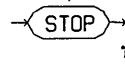
26 *nextstate* (Basic SDL 2.6.7.2.1)



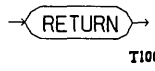
27 *join* (Basic SDL 2.6.7.2.2)



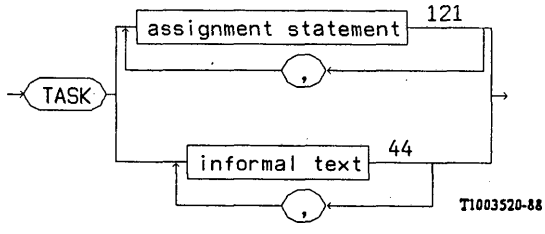
28 *stop* (Basic SDL 2.6.7.2.3)



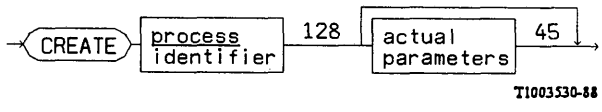
29 *return* (Basic SDL 2.6.7.2.4)



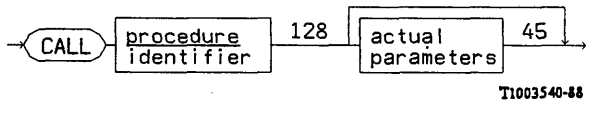
30 *task* (Basic SDL 2.7.1)



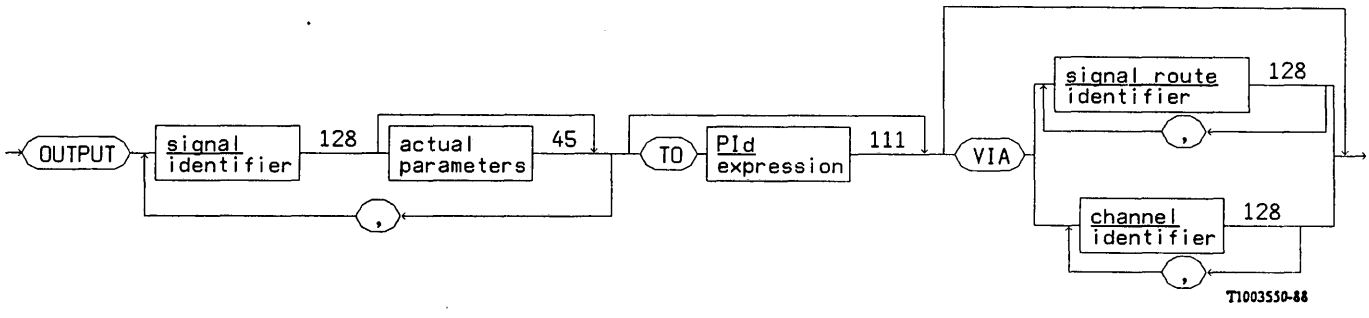
31 *create request* (Basic SDL 2.7.2)



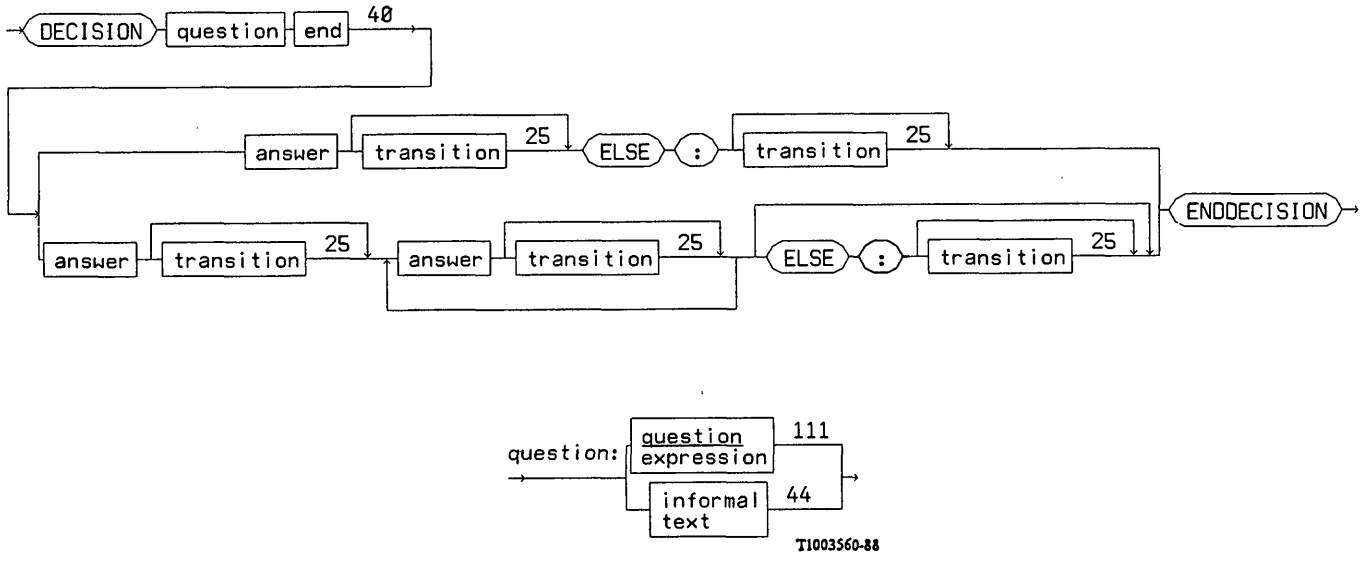
32 *procedure call* (Basic SDL 2.7.3)



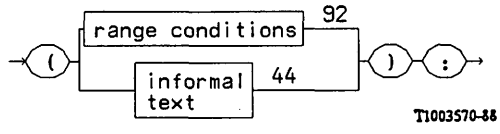
33 *output* (Basic SDL 2.7.4)



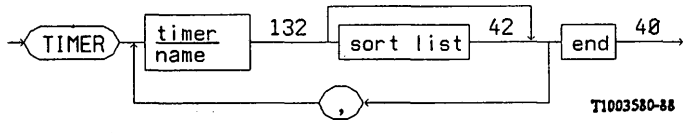
34 *decision* (Basic SDL 2.7.5)



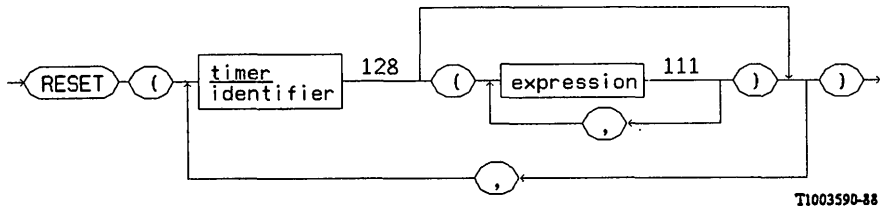
35 *answer* (Basic SDL 2.7.5)



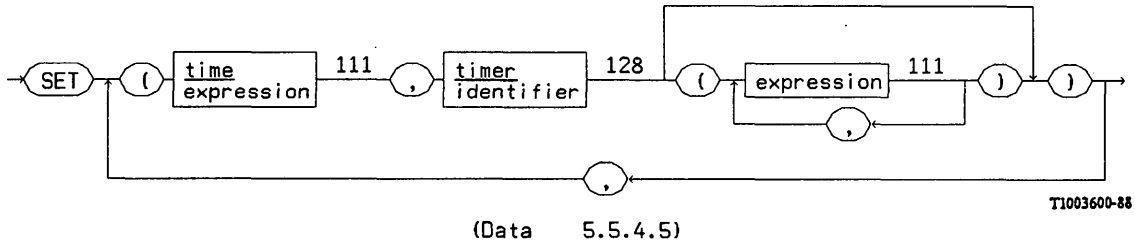
36 *timer definition* (Basic SDL 2.8)



37 *reset* (Basic SDL 2.8)

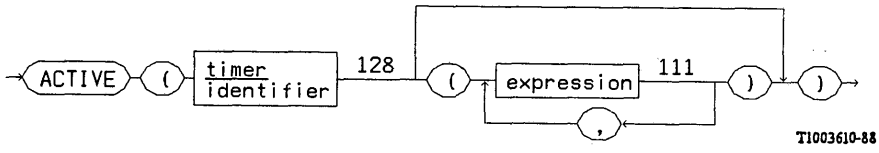


38 *set* (Basic SDL 2.8)

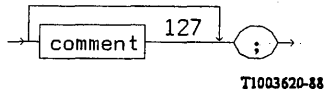


(Data 5.5.4.5)

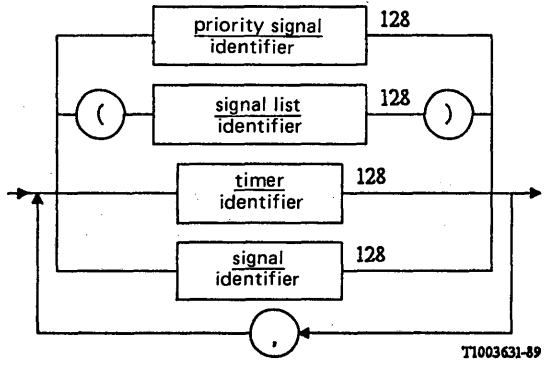
39 *timer active expression* (Data 5.5.4.5)



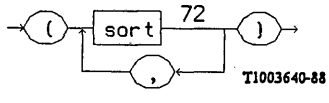
40 *end*



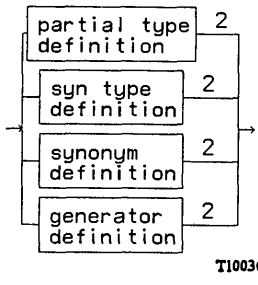
41 *signal list*



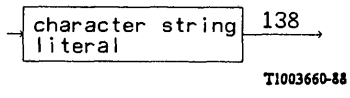
42 *sort list*



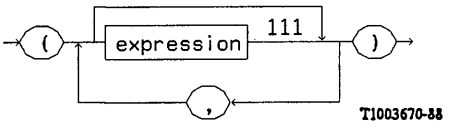
43 *data definition*



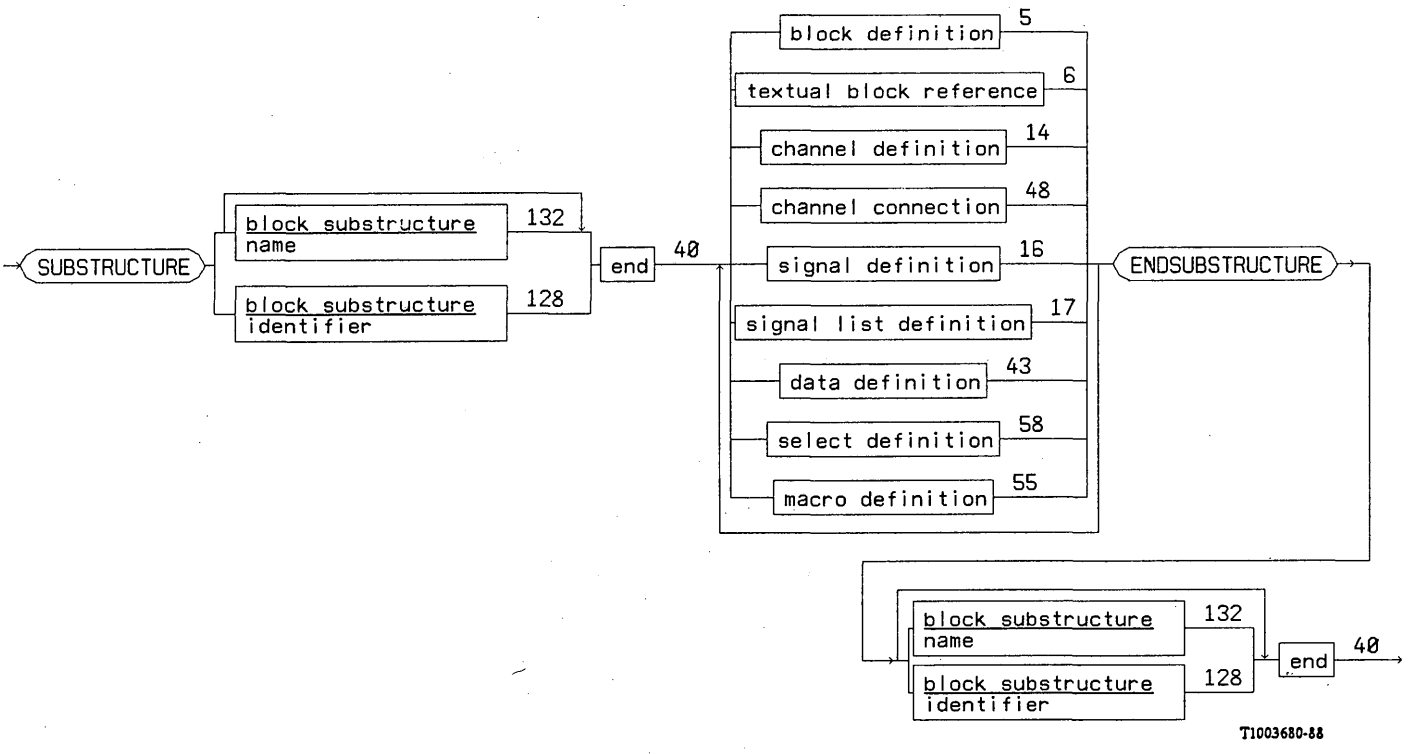
44 *informal text*



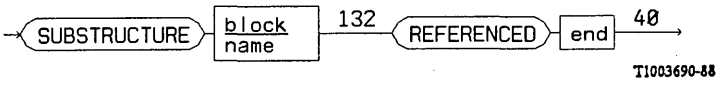
45 *actual parameters*



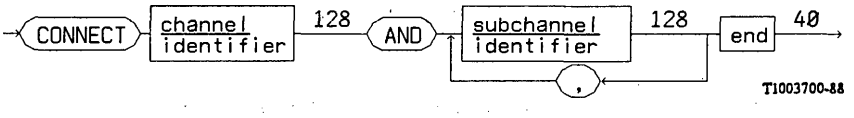
46 *block substructure definition* (Structural concepts 3.2.2)



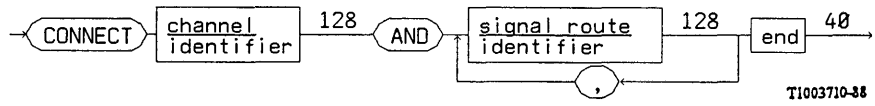
47 *block substructure reference* (Structural concepts 3.2.2)



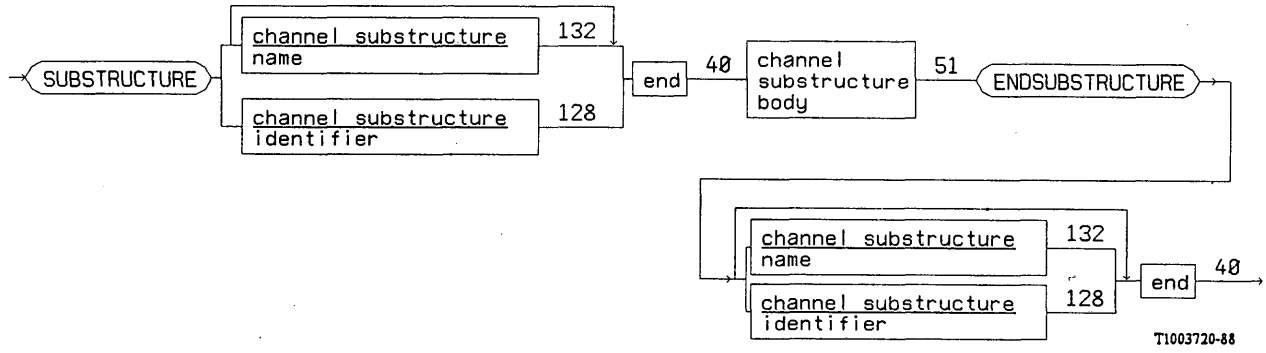
48 *channel connection* (Structural concepts 3.2.2)



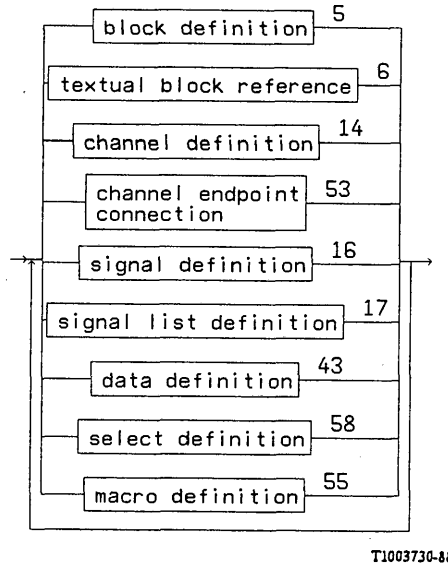
49 *channel to route connection* (Structural concepts 3.2.2)



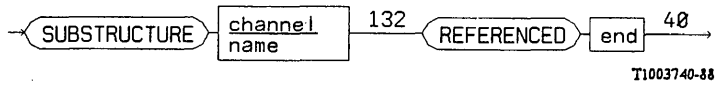
50 *channel substructure definition* (Structural concepts 3.2.3)



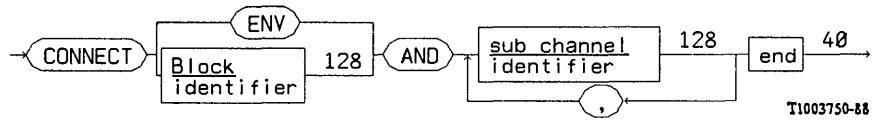
51 *channel substructure body* (Structural concepts 3.2.3)



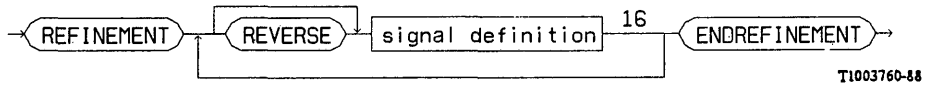
52 *channel substructure reference* (Basic SDL 2.5.1)



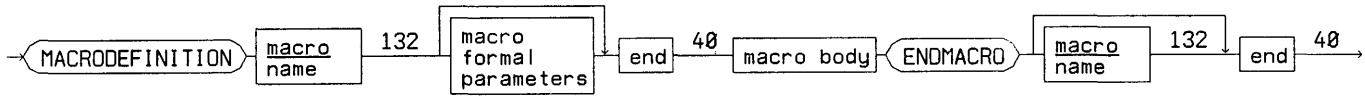
53 *channel endpoint connection* (Structural concepts 3.2.3)

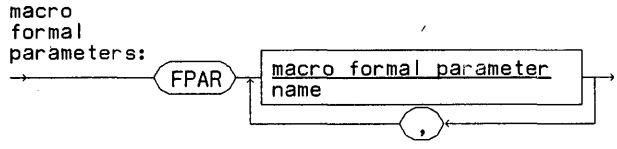


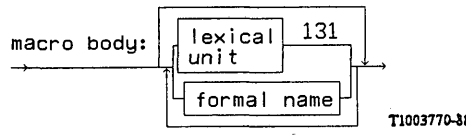
54 *signal refinement* (Structural concepts 3.3)



55 *macro definition* (Additional concepts 4.2.2)

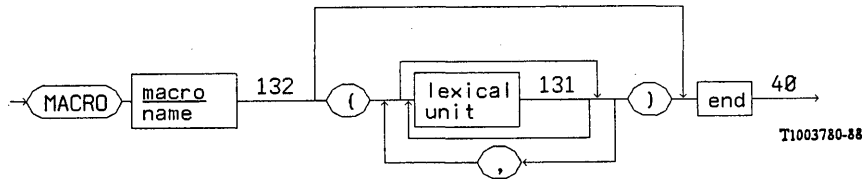




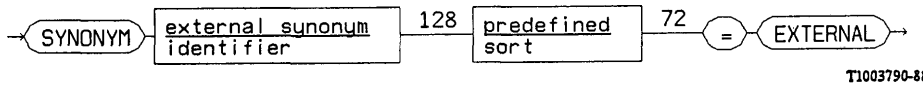


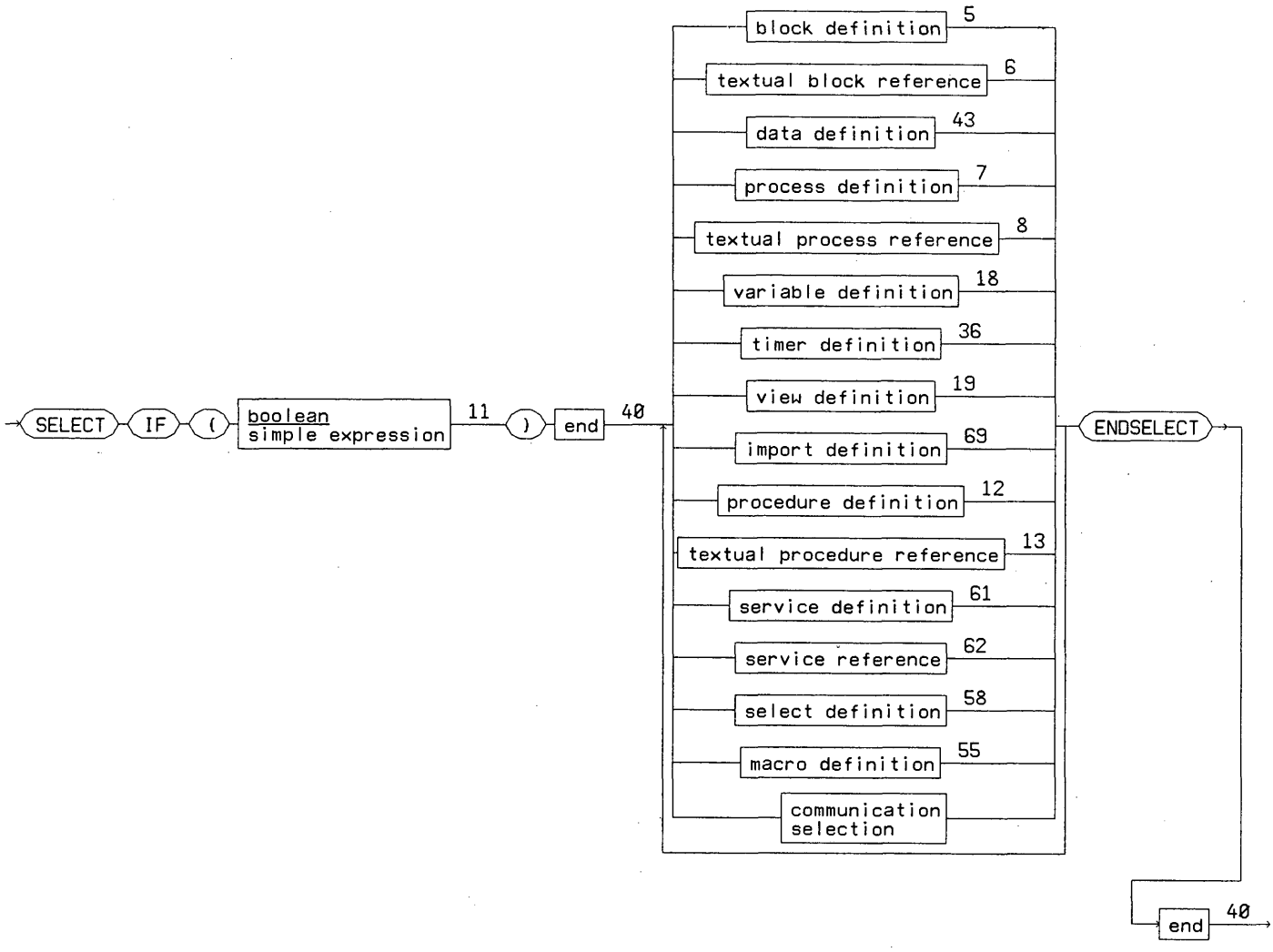
(Additional concepts 4.2.3)

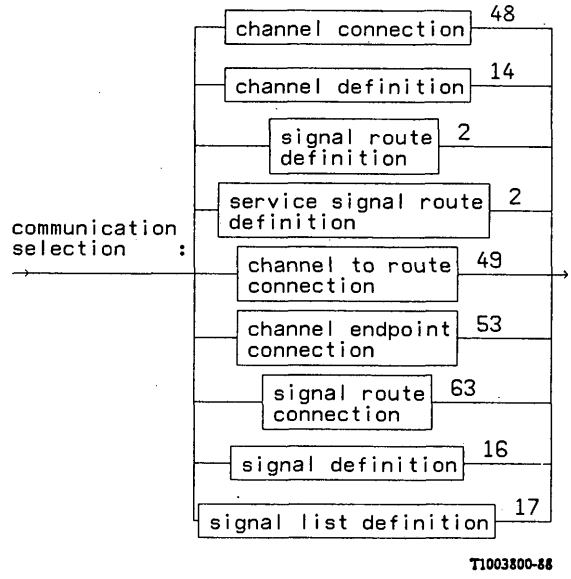
56 *macro call* (Additional concepts 4.2.3)



57 *external synonym definition* (Additional concepts 4.3.1)

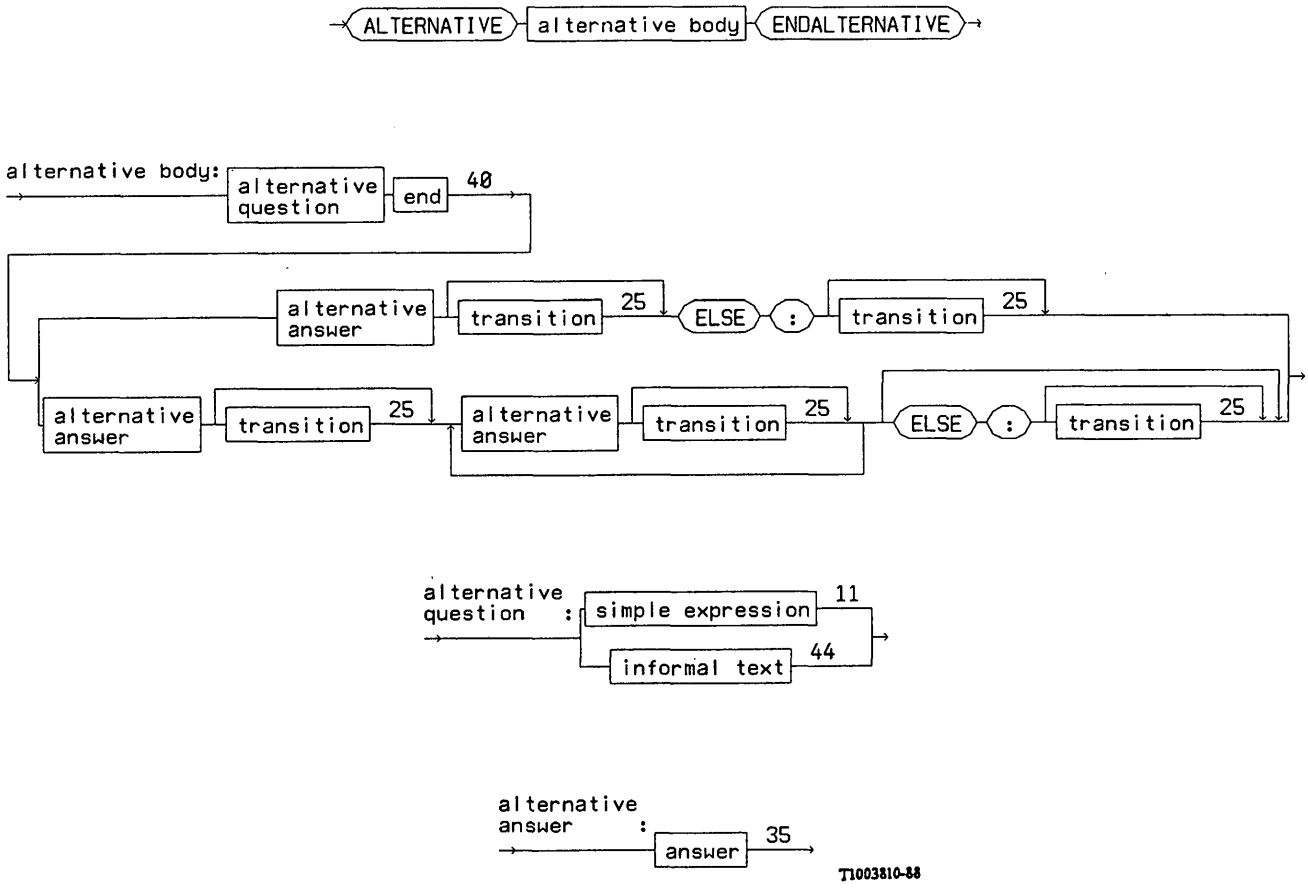



SELECT
IF
(
boolean
simple expression
11
)
end
40
block definition
5
textual block reference
6
data definition
43
process definition
7
textual process reference
8
variable definition
18
timer definition
36
view definition
19
import definition
69
procedure definition
12
textual procedure reference
13
service definition
61
service reference
62
select definition
58
macro definition
55
communication selection
ENDSELECT
end
40

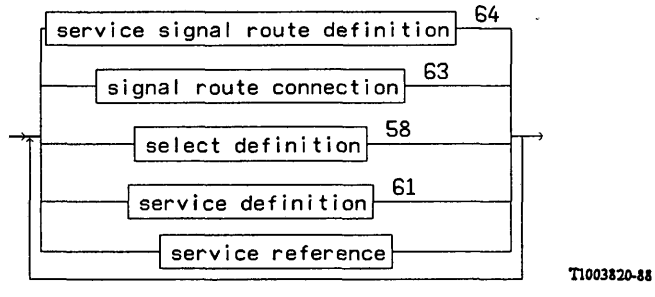

communication selection :
channel connection
48
channel definition
14
signal route definition
2
service signal route definition
2
channel to route connection
49
channel endpoint connection
53
signal route connection
63
signal definition
16
signal list definition
17

T1003800-88

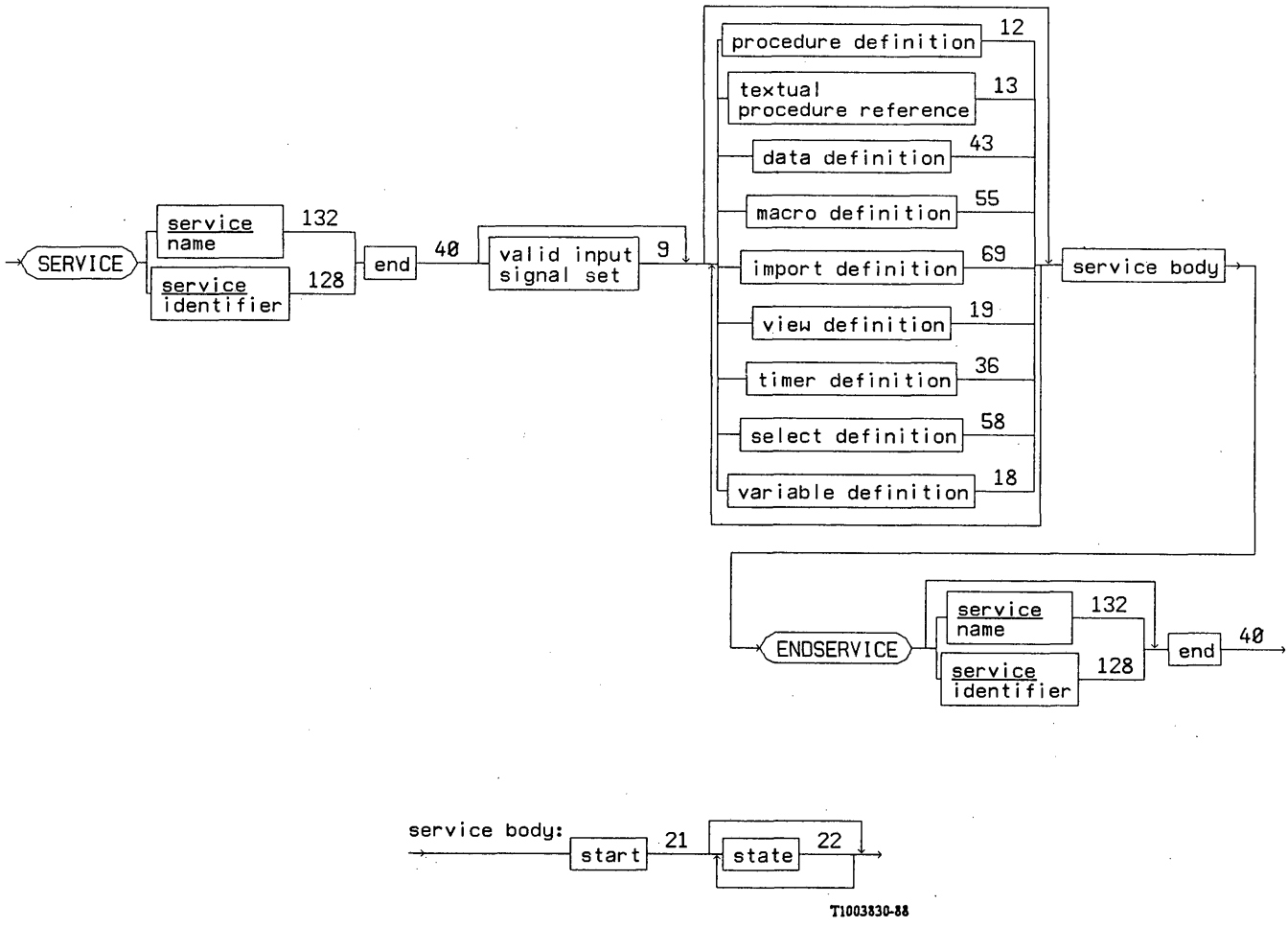
59 *transition option* (Additional concepts 4.3.4)



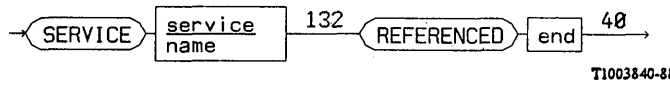
60 *service decomposition* (Additional concepts 4.10.1)



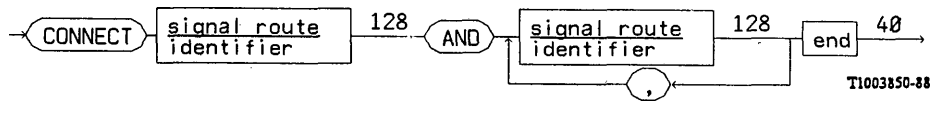
61 *service definition* (Additional concepts 4.10.2)



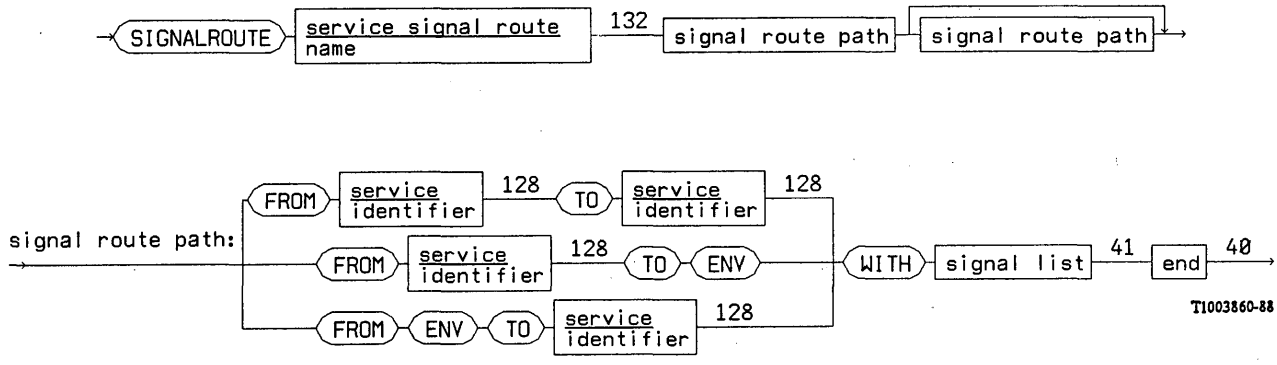
62 *service reference* (Additional concepts 4.10.1)



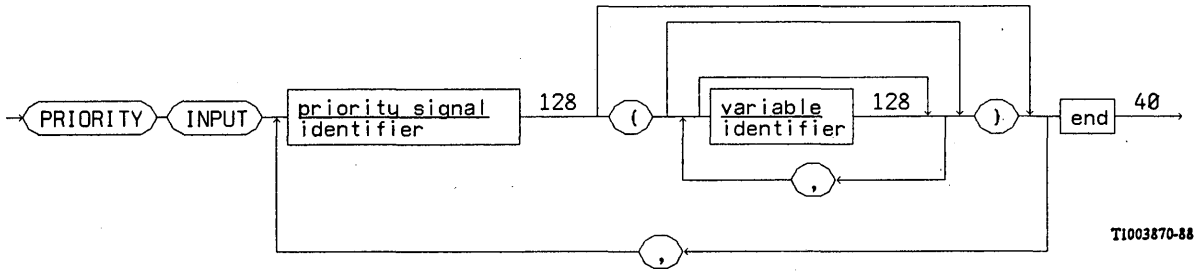
63 *signal route connection* (Additional concepts 4.10.1)



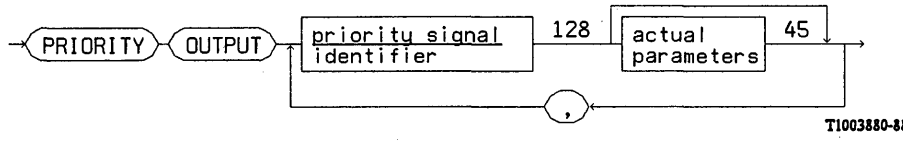
64 *service signal definition* (Additional concepts 4.10.1)



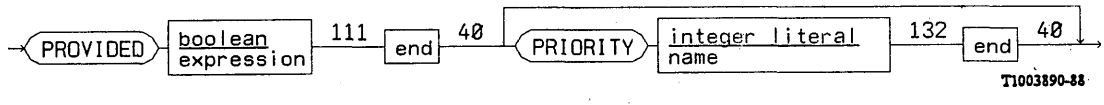
65 *priority input* (Additional concepts 4.10.2)



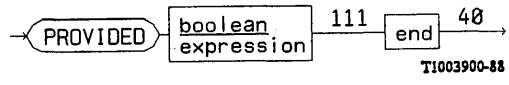
66 *priority output* (Additional concepts 4.10.2)



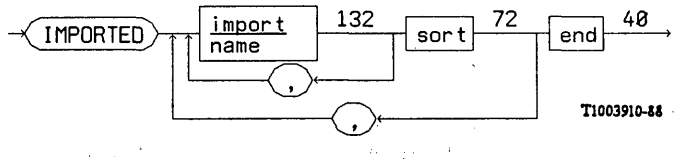
67 *continuous signal* (Additional concepts 4.11)



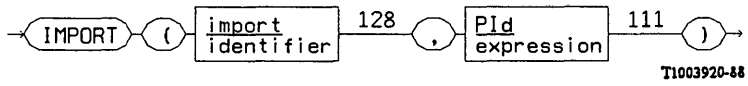
68 *enabling condition* (Additional concepts 4.12)



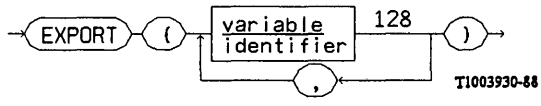
69 *import definition* (Additional concepts 4.13)



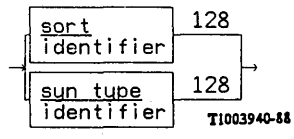
70 *import expression* (Additional concepts 4.13)



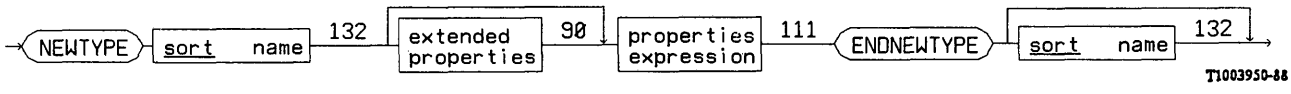
71 *export* (Additional concepts 4.13)



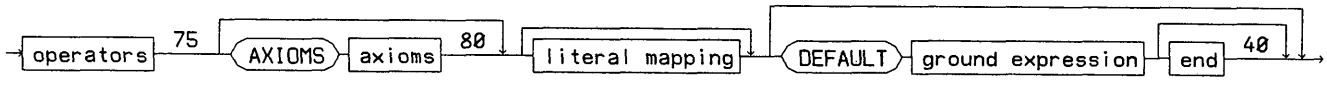
72 *sort* (Data 5.2.2)

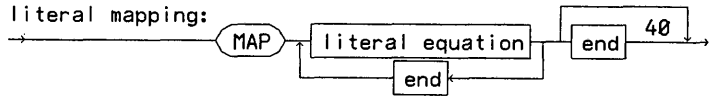


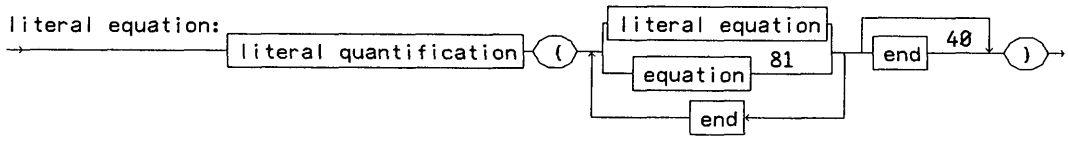
73 *partial type definition* (Data 5.2.1)

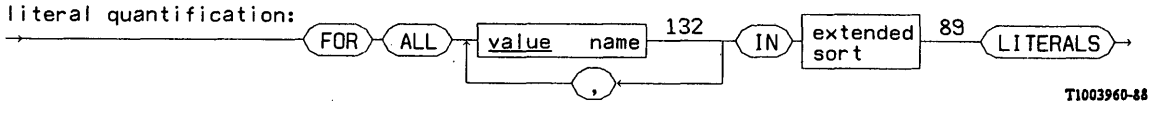


74 *properties expression* (Data 5.2.1 and 5.5.3.3)

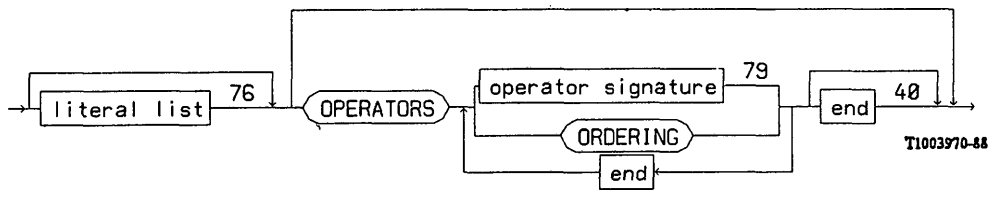




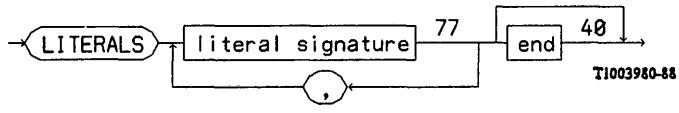




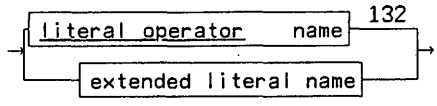
75 *operators* (Data 5.2.2 and 5.4.1.8)

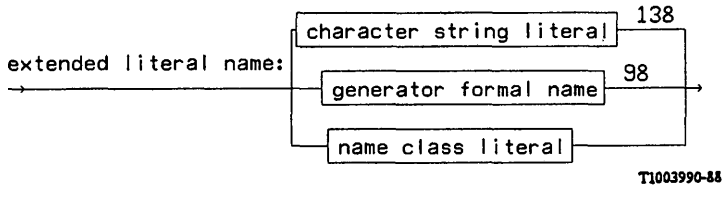


76 *literal list* (Data 5.2.2)

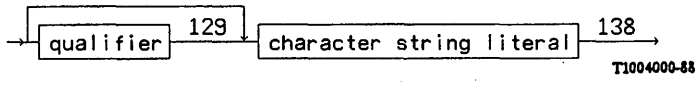


77 *literal signature* (Data 5.2.2 & 5.4.1.8)

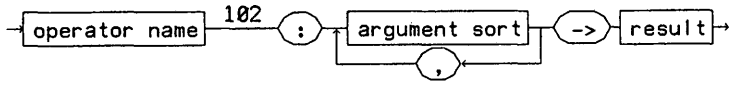


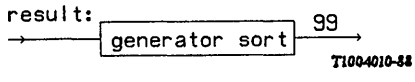


78 *character string literal identifier*

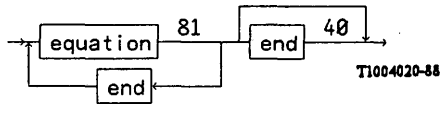


79 *operator signature* (Data 5.2.2)

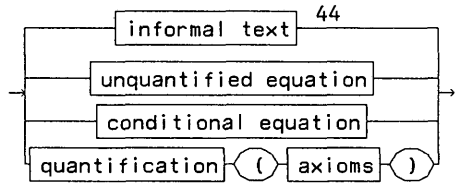


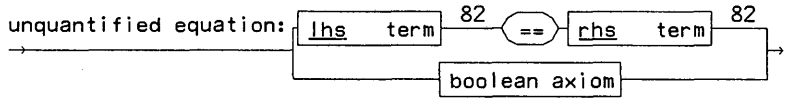


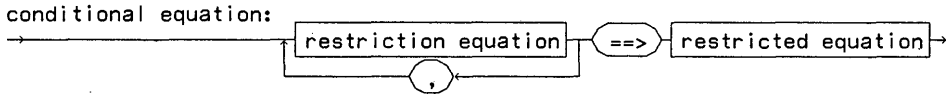
80 *axioms* (Data 5.2.3 and 5.2.4)

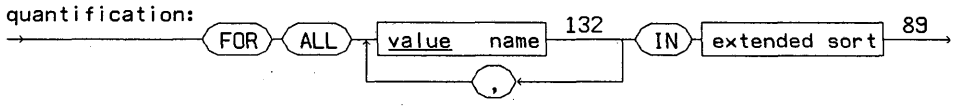


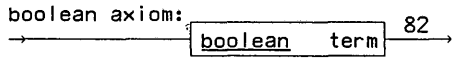
81 *equation*

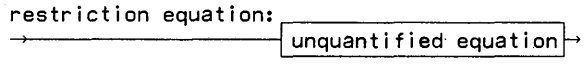


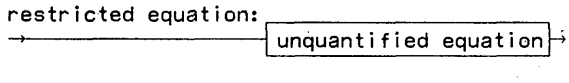






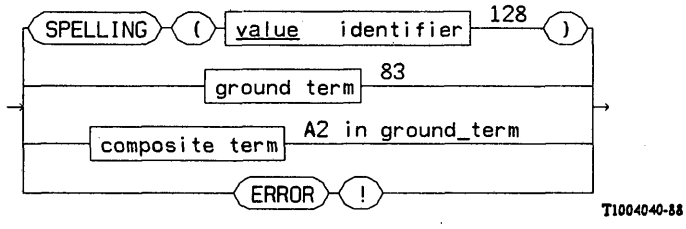




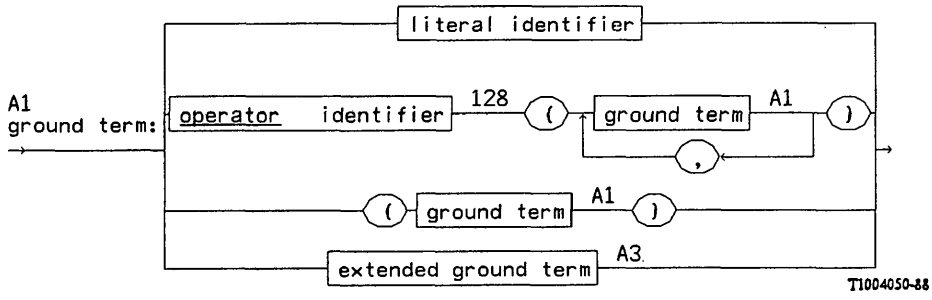


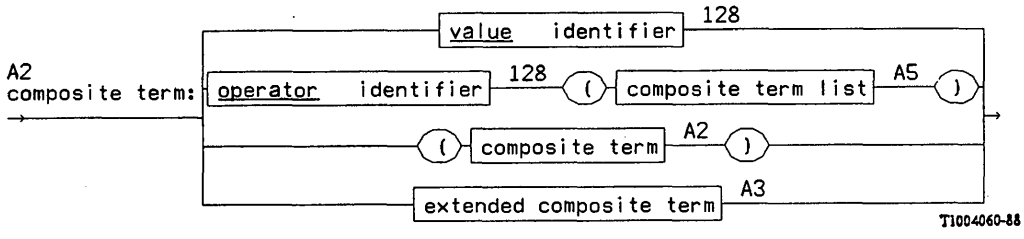
T1004030-88

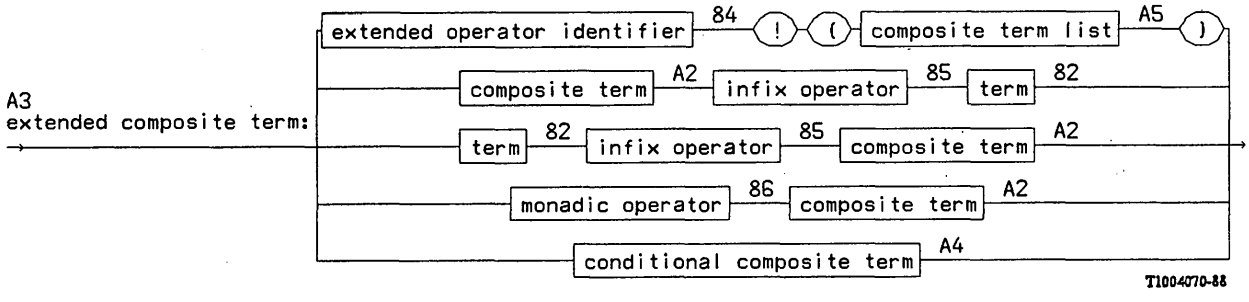
82 *term*

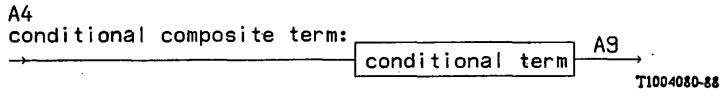


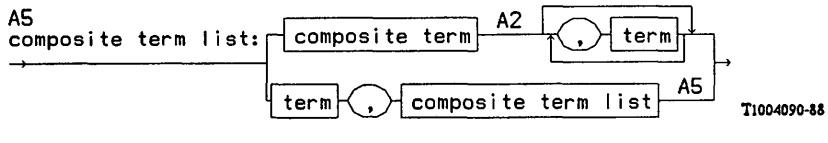
T1004040-88

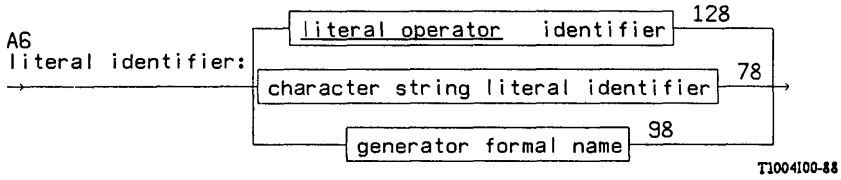

literal identifier
A1
ground term:
operator identifier
128
(
ground term
A1
)
,
(
ground term
A1
)
extended ground term
A3
T1004050-88

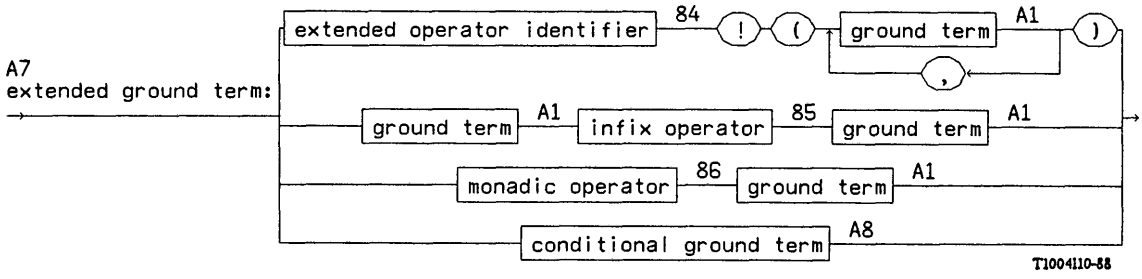

value identifier
128
A2
composite term:
operator identifier
128
(
composite term list
A5
)
(
composite term
A2
)
extended composite term
A3
T1004060-88

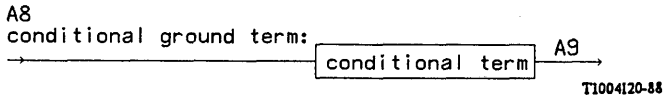

extended operator identifier
84
!
(
composite term list
A5
)
composite term
A2
infix operator
85
term
82
A3
extended composite term:
term
82
infix operator
85
composite term
A2
monadic operator
86
composite term
A2
conditional composite term
A4
T1004070-88

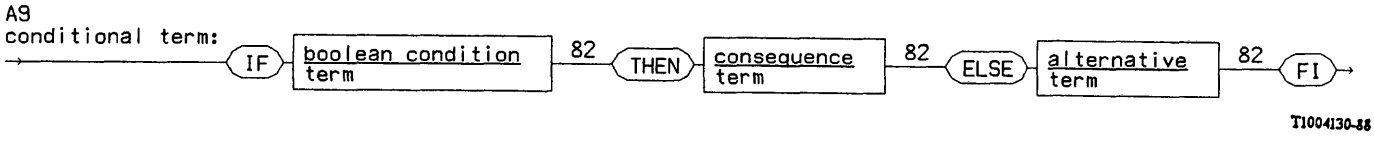

A4
conditional composite term:
conditional term
A9
T1004080-88


A5
composite term list:
composite term
A2
,
term
term
,
composite term list
A5
T1004090-88

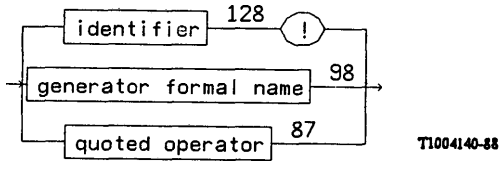




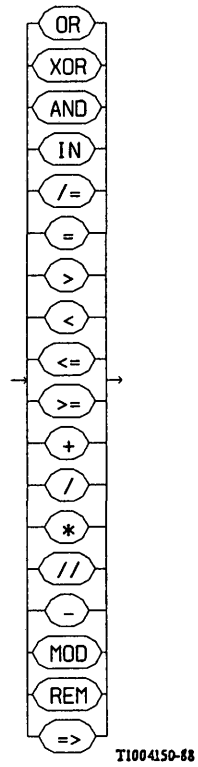




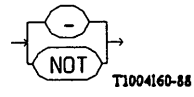
84 *extended operator identifier*



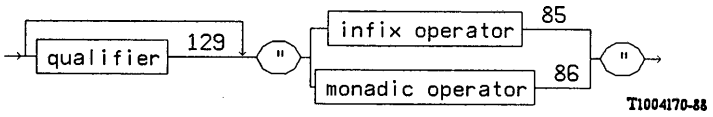
85 *infix operator*



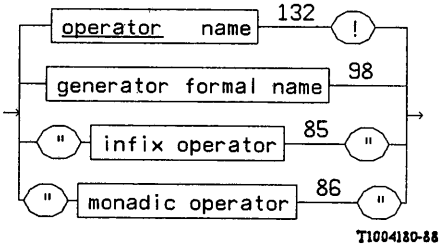
86 *monadic operator*



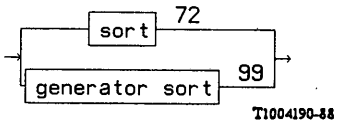
87 *quoted operator*



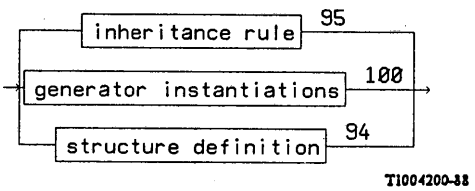
88 *extended operator name* (Data 5.4.1)



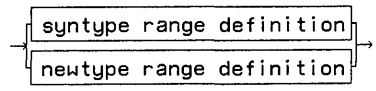
89 *extended sort* (Data 5.4.1.9)

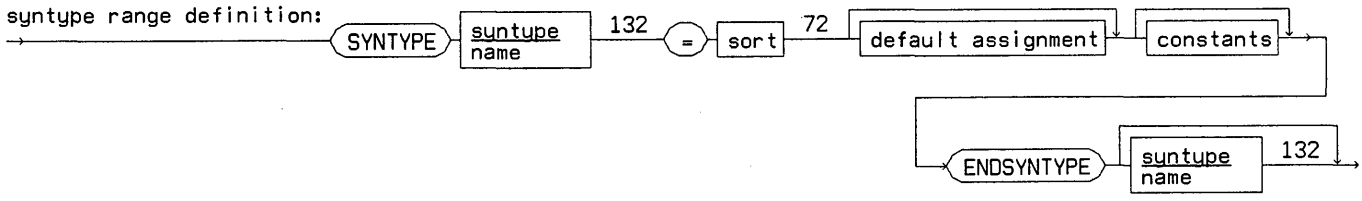


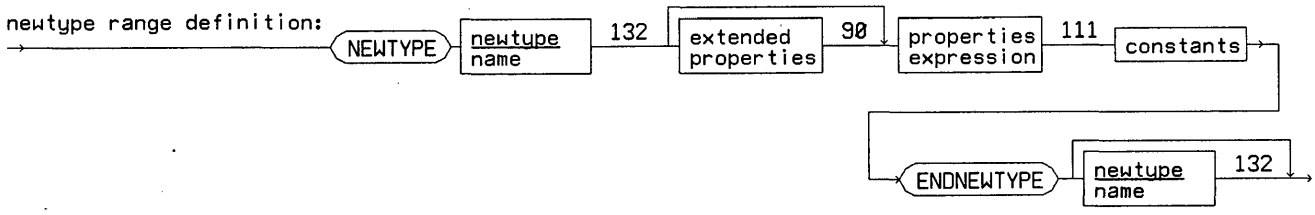
90 *extended properties* (Data 5.4.1)

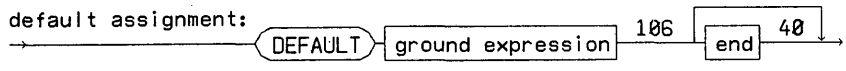


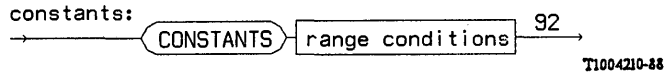
91 *syntype definition* (Data 5.4.1.9)



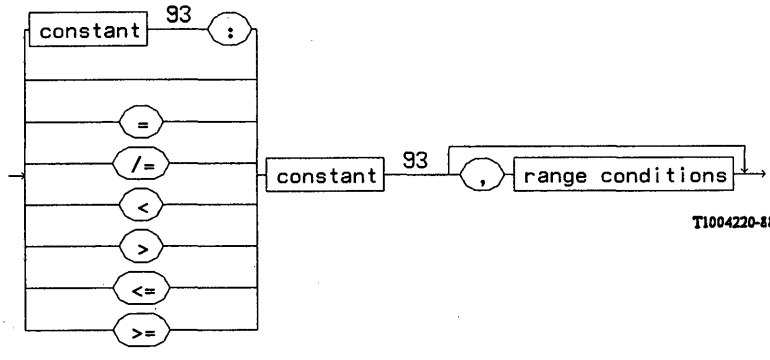




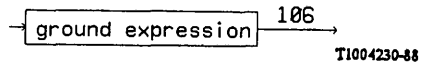




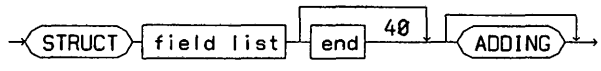
92 *range conditions* (Data 5.4.1.9.1)

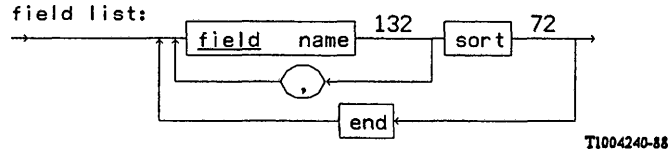


93 *constant* (Data 5.4.1.9)

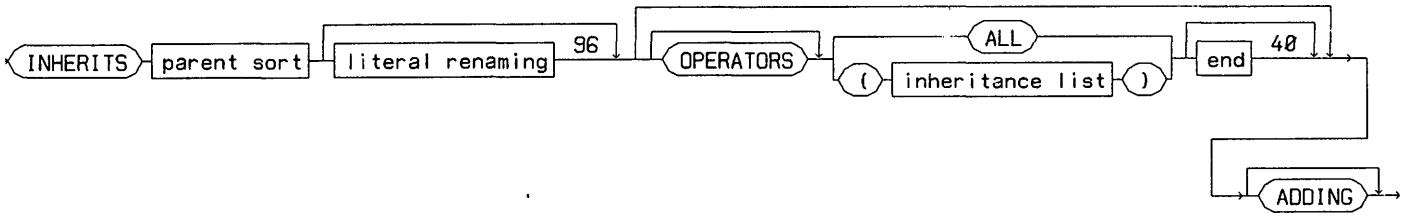


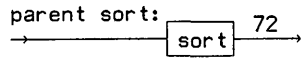
94 *structure definition* (Data 5.4.1.10)

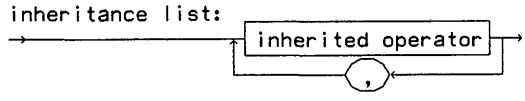


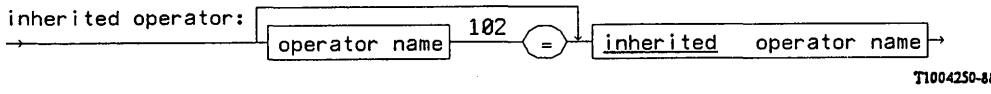


95 *inheritance rule* (Data 5.4.1.11)



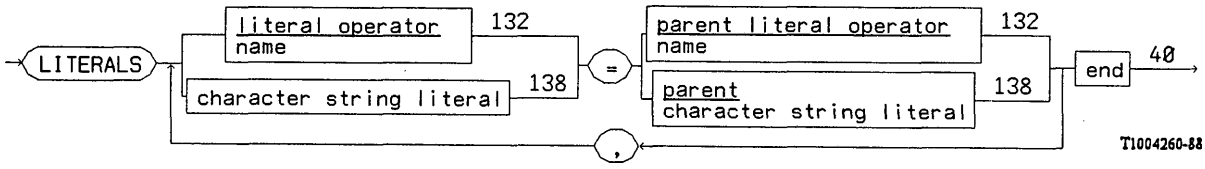




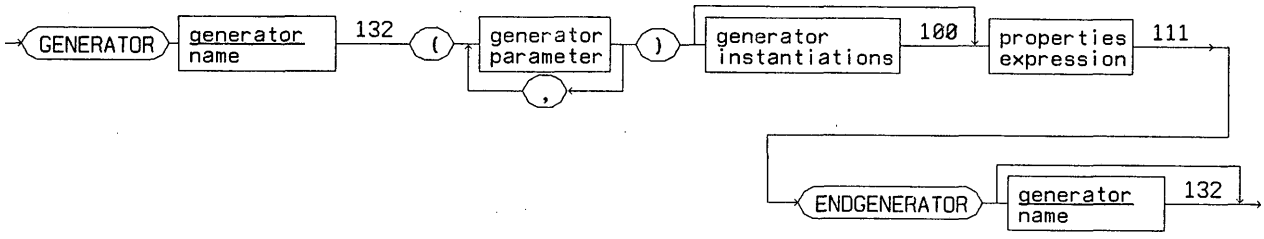


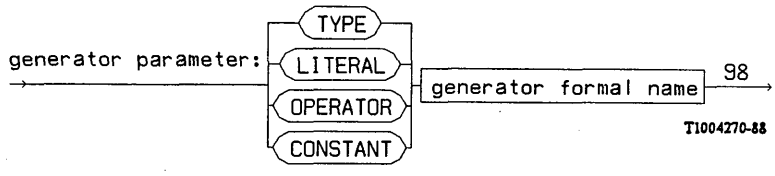
T1004250-88

96 *literal renaming* (Data 5.4.1.11)

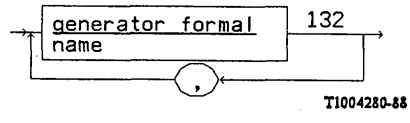


97 *generator definition* (Data 5.4.1.12.1)

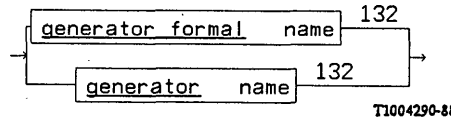




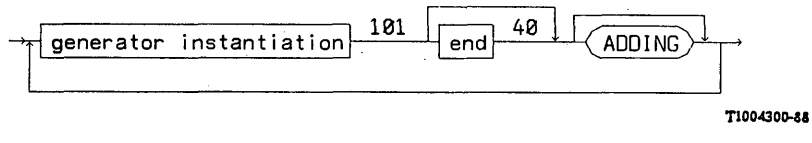
98 *generator formal name* (Data 5.4.1.12.1)



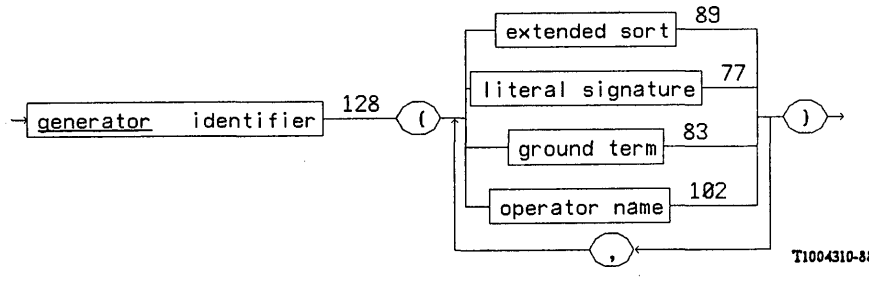
99 *generator sort* (Data 5.4.1.12.1)



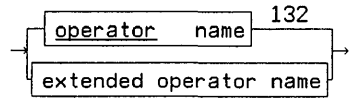
100 *generator instantiations* (Data 5.4.1.12.2)

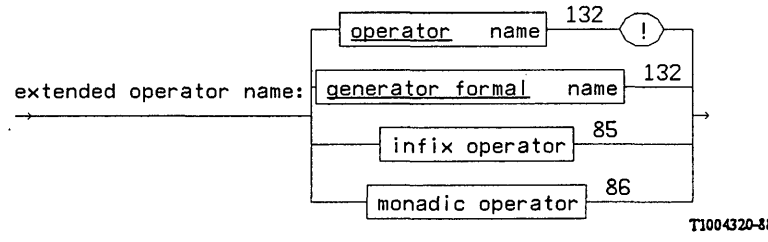


101 *generator instantiation* (Data 5.4.1.12.2)

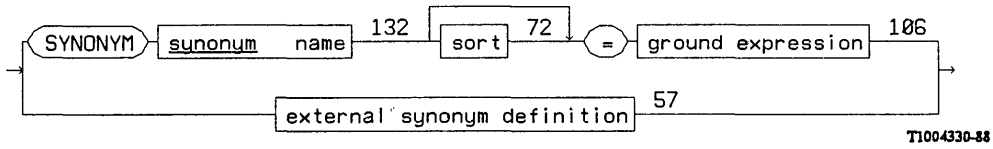


102 *operator name*

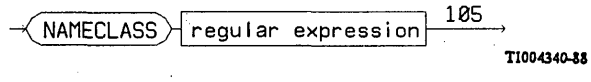




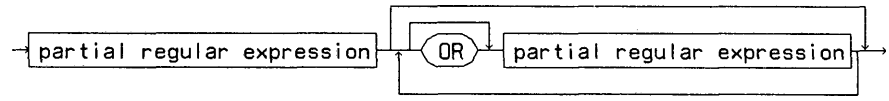
103 *synonym definition* (Data 5.4.1.13)

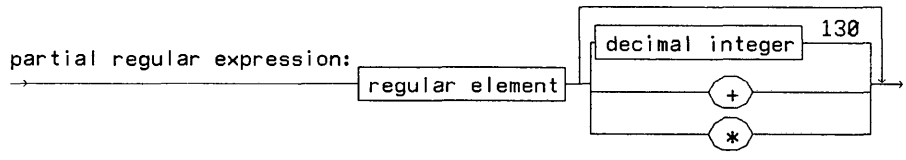


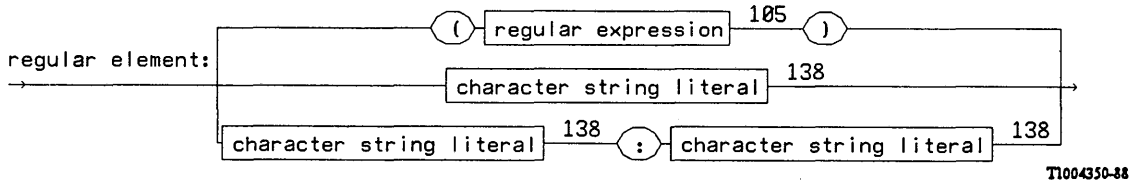
104 *name class literal* (Data 5.4.1.14)



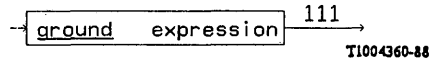
105 *regular expression* (Data 5.4.1.14)



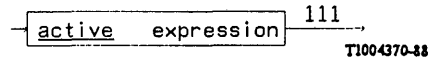




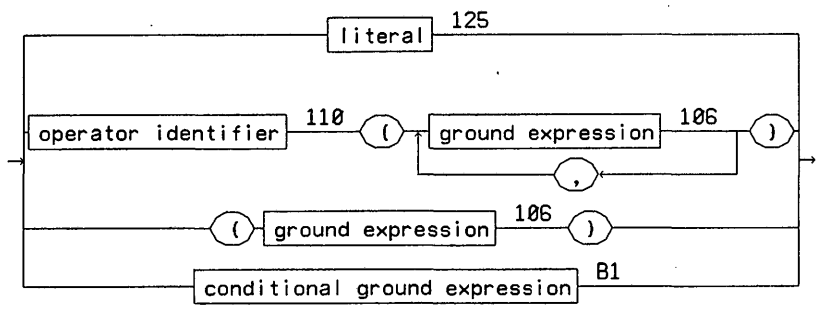
106 *ground expression* (Data 5.4.2.1)

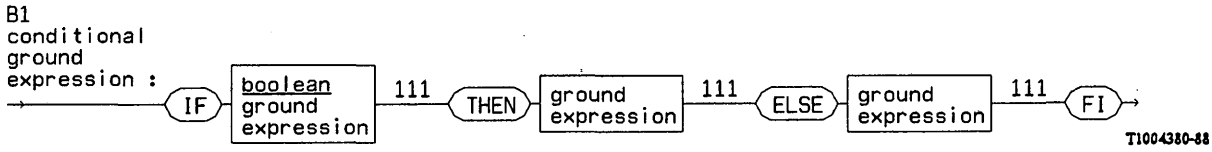


107 *active expression* (Data 5.4.2.2)

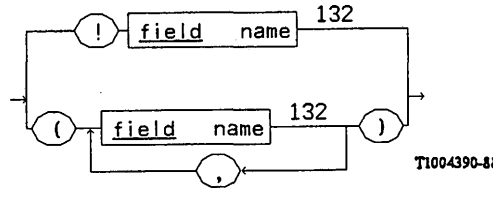


108 *ground primary* (Data 5.4.2.2)

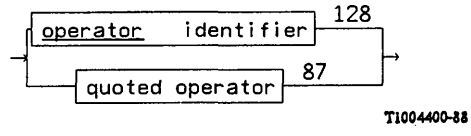




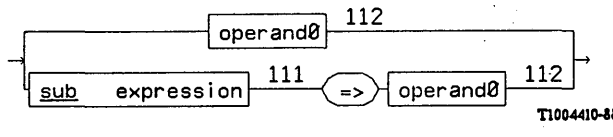
109 *field selection*



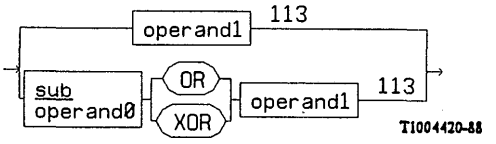
110 *operator identifier*



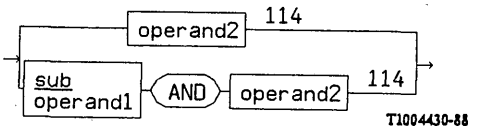
111 *expression* (Data 5.5.2.1)



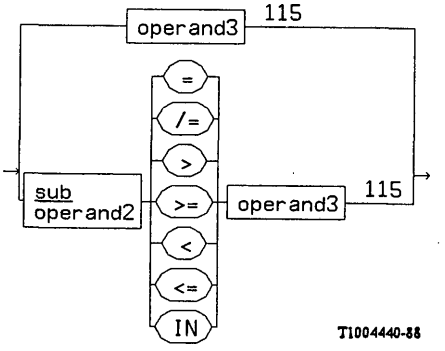
112 *operand0* (Data 5.5.2.1)



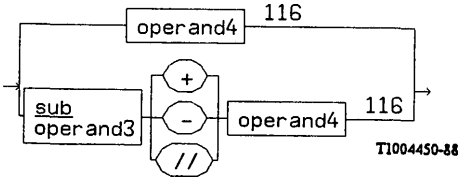
113 *operand1* (Data 5.5.2.1)



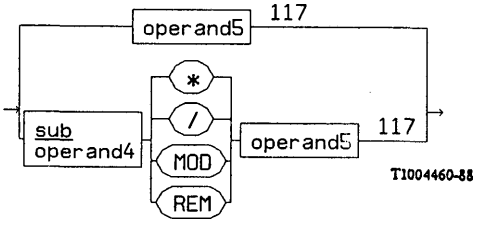
114 *operand2* (Data 5.5.2.1)



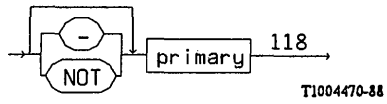
115 *operand3* (Data 5.5.2.1)



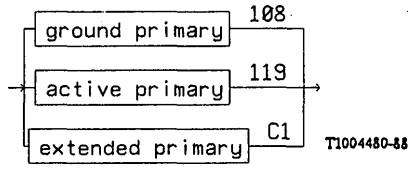
116 *operand4* (Data 5.5.2.1)



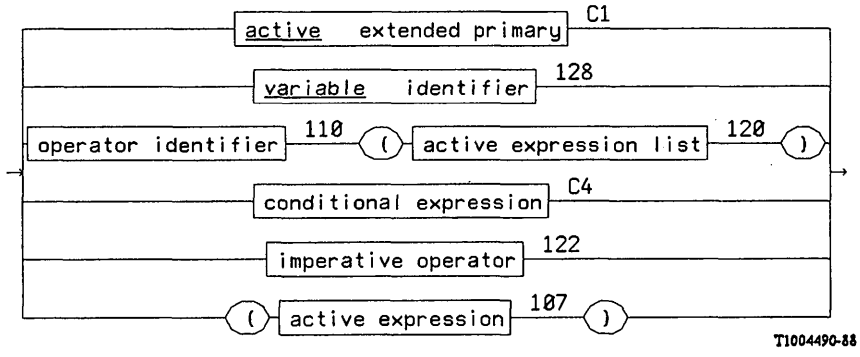
117 *operand5* (Data 5.5.2.1)



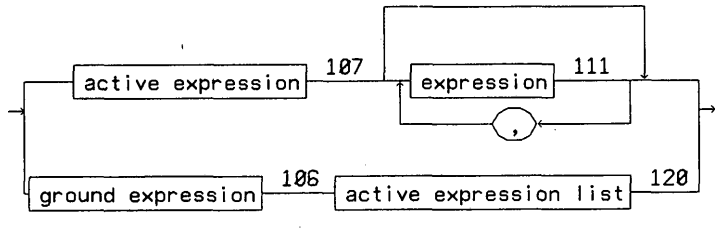
118 *primary*

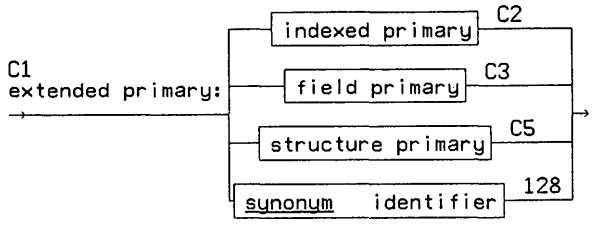


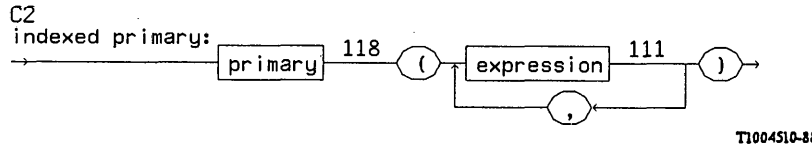
119 *active primary*

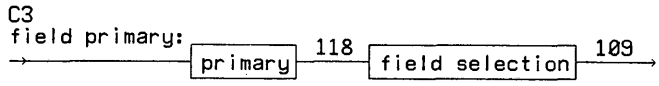


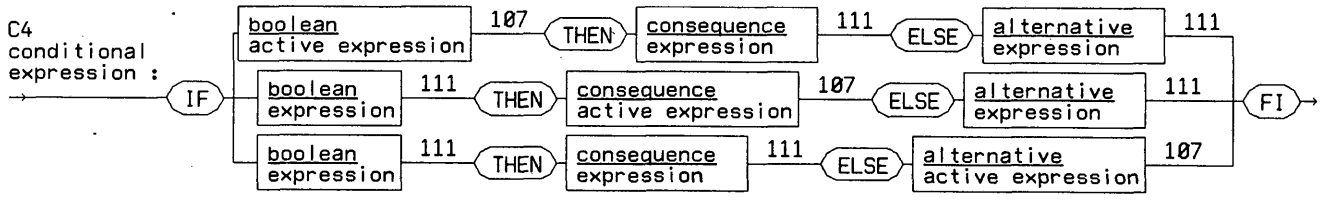
120 *active expression list* (Data 5.5.2.1)

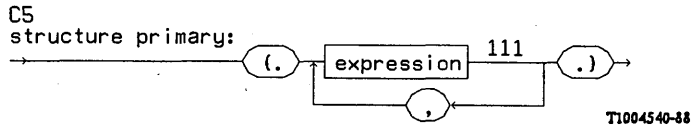




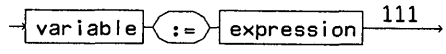


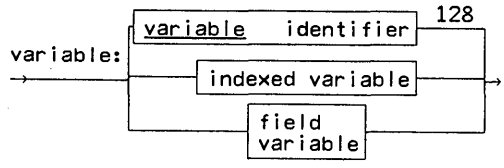


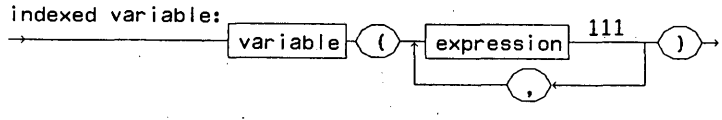


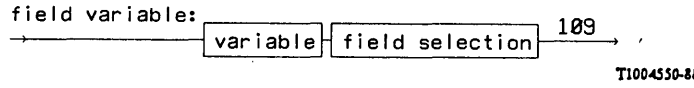


121 *assignment statement*

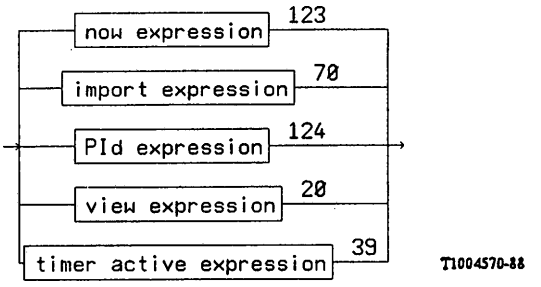




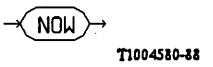




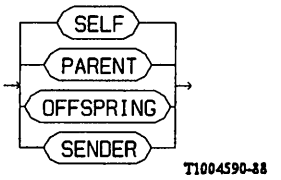
122 *imperative operator* (Data 5.5.4)



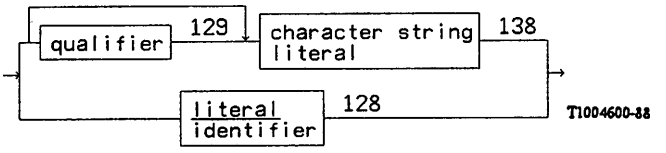
123 *now expression*



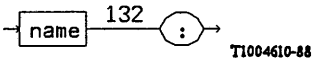
124 *PId expression*



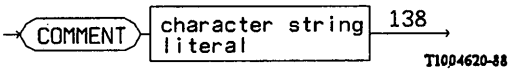
125 *literal*



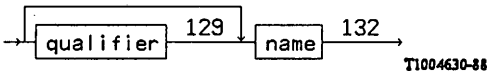
126 *label*



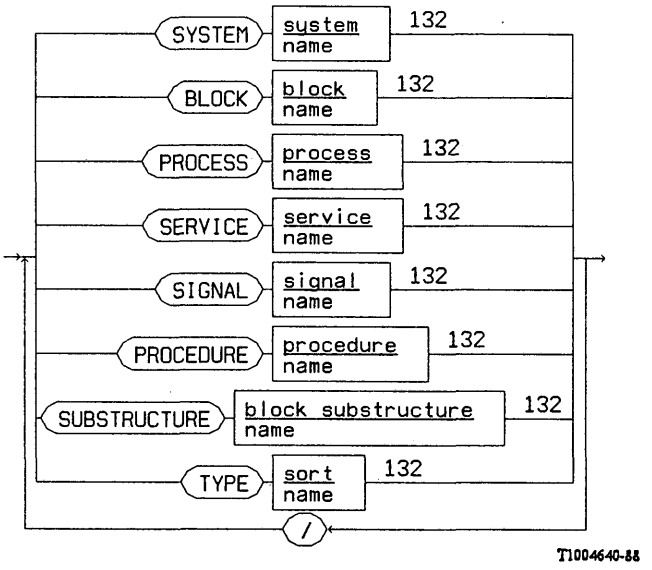
127 *comment*



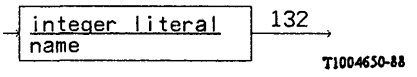
128 *identifier*



129 *qualifier*

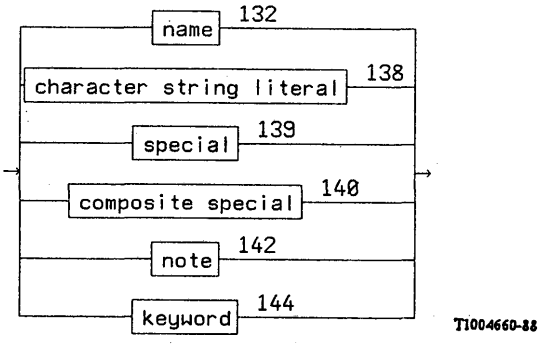


130 *decimal integer*

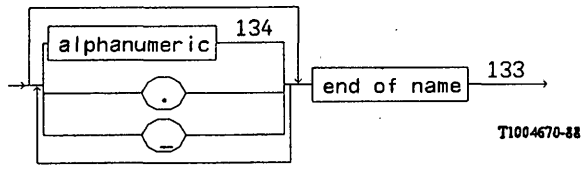


Lexical rules syntax diagrams

131 *lexical unit*

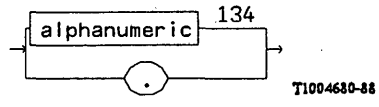


132 *name*

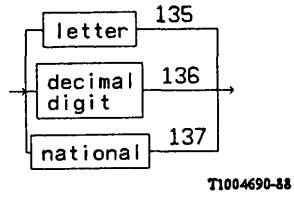


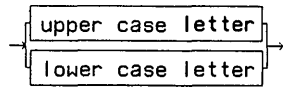
A *name* must contain at least one alphabetic character.

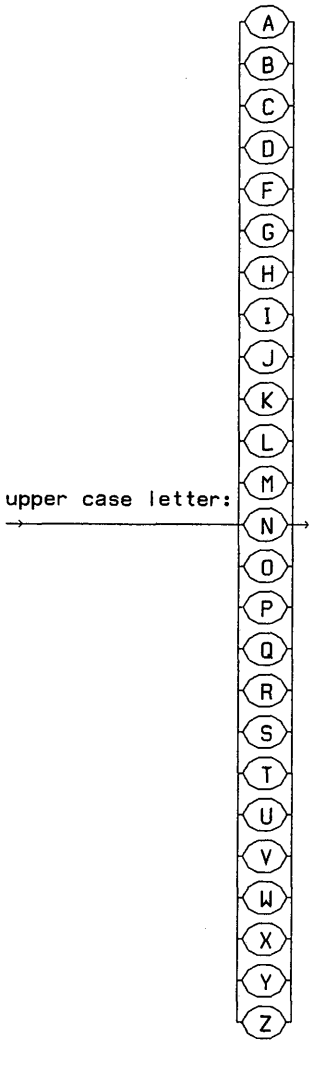
133 *end of name*

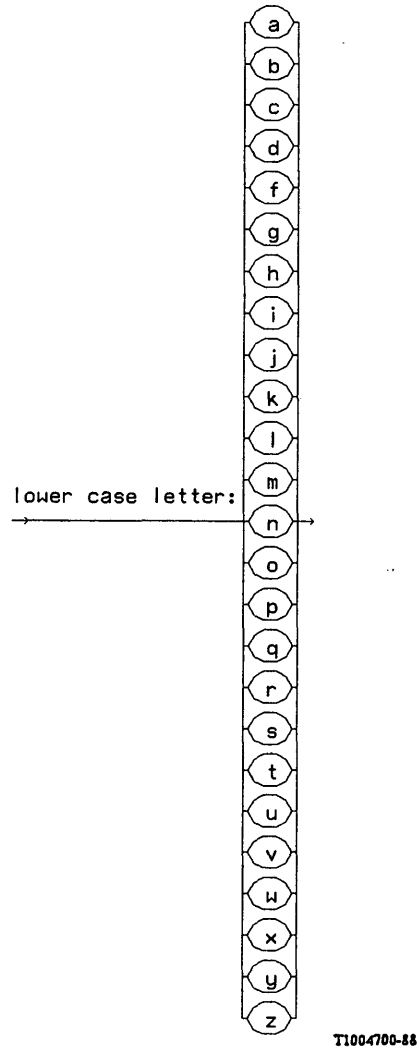


134 *alphanumeric*

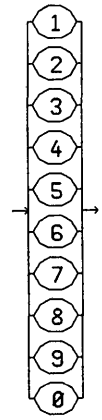



upper case letter
lower case letter


upper case letter:
A
B
C
D
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z
T1004700-88

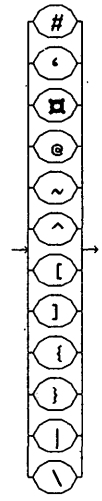

lower case letter:
a
b
c
d
f
g
h
i
j
k
l
m
n
o
p
q
r
s
t
u
v
w
x
y
z
T1004700-88

136 *decimal digit*



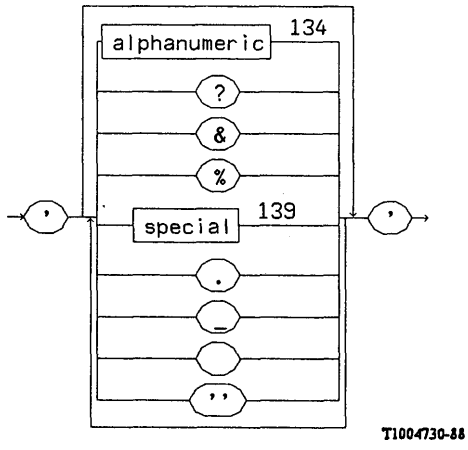
T1004710-88

137 *national*

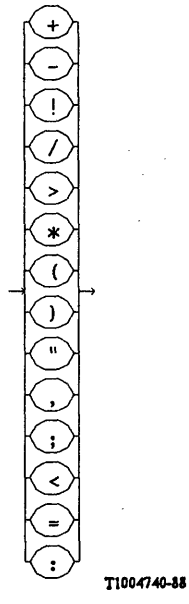


T1004720-88

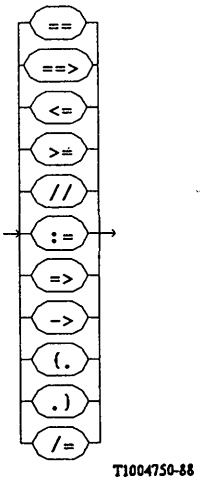
138 *character string literal*



139 *special*

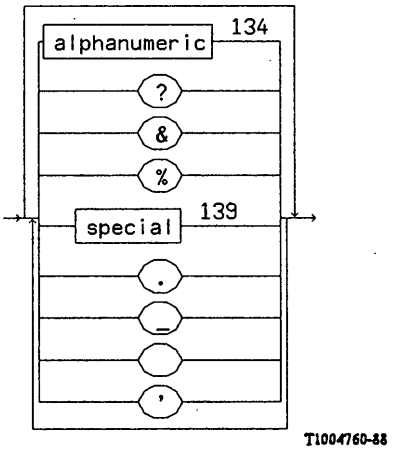


140 *composite special*



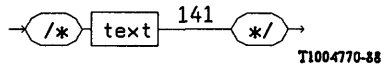
T1004750-88

141 *text*

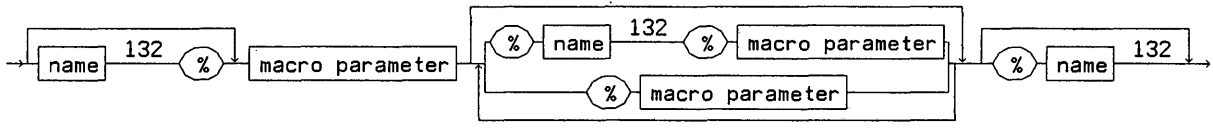


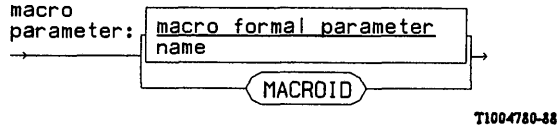
T1004760-88

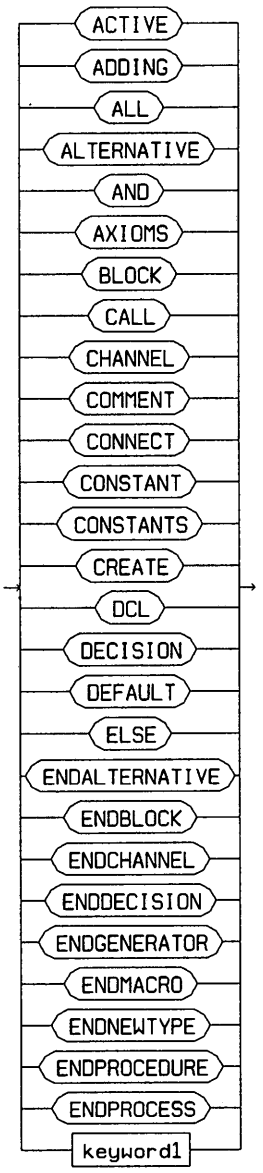
142 *note*

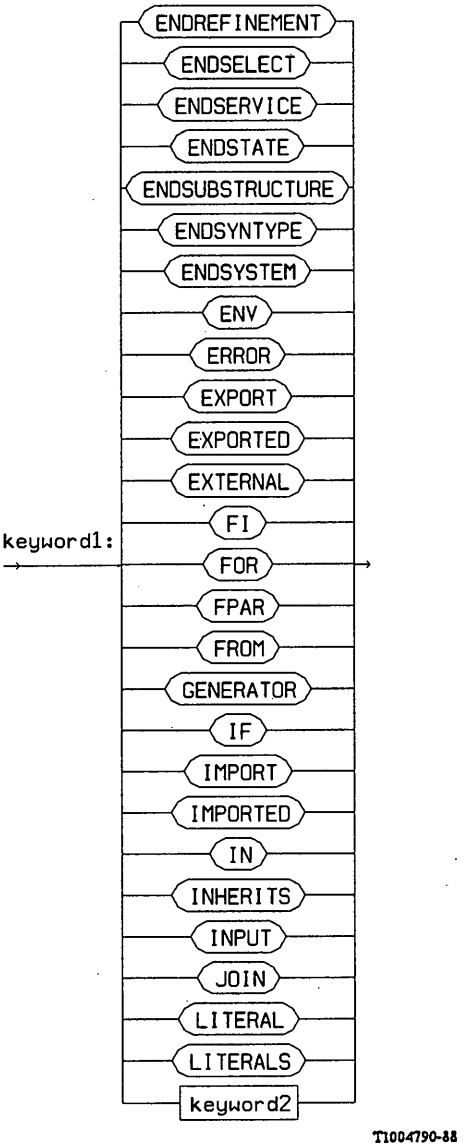


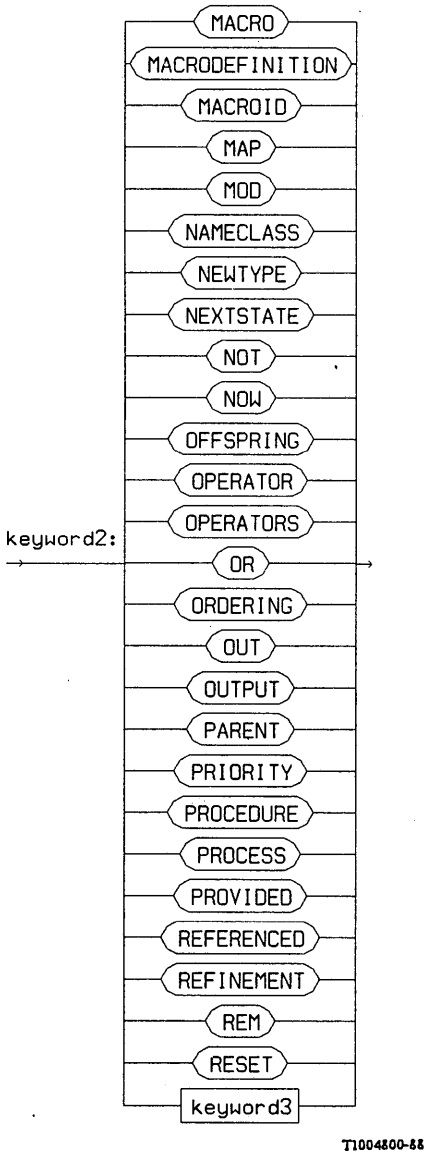
143 *formal name*

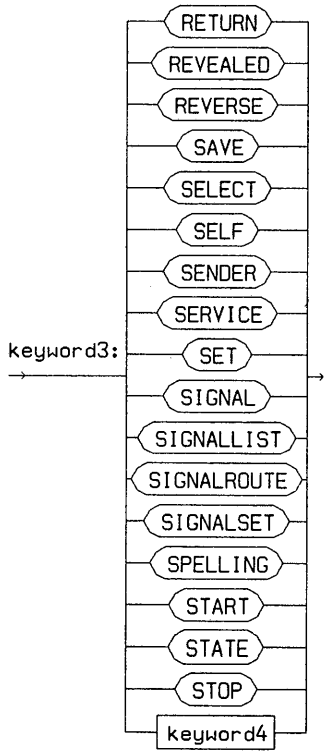


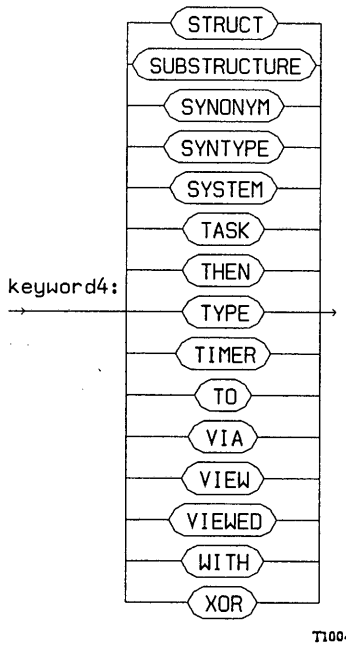



ACTIVE
ADDING
ALL
ALTERNATIVE
AND
AXIOMS
BLOCK
CALL
CHANNEL
COMMENT
CONNECT
CONSTANT
CONSTANTS
CREATE
DCL
DECISION
DEFAULT
ELSE
ENDALTERNATIVE
ENDBLOCK
ENDCHANNEL
ENDDECISION
ENDGENERATOR
ENDMACRO
ENDNEWTYPE
ENDPROCEDURE
ENDPROCESS
keyword1
T1004790-88


keyword1:
ENDREFINEMENT
ENDSELECT
ENDSERVICE
ENDSTATE
ENDSUBSTRUCTURE
ENDSYNTYPE
ENDSYSTEM
ENV
ERROR
EXPORT
EXPORTED
EXTERNAL
FI
FOR
FPAR
FROM
GENERATOR
IF
IMPORT
IMPORTED
IN
INHERITS
INPUT
JOIN
LITERAL
LITERALS
keyword2
T1004790-88


keyword2:
MACRO
MACRODEFINITION
MACROID
MAP
MOD
NAMECLASS
NEWTYPE
NEXTSTATE
NOT
NOW
OFFSPRING
OPERATOR
OPERATORS
OR
ORDERING
OUT
OUTPUT
PARENT
PRIORITY
PROCEDURE
PROCESS
PROVIDED
REFERENCED
REFINEMENT
REM
RESET
keyword3
T1004600-88


keyword3:
RETURN
REVEALED
REVERSE
SAVE
SELECT
SELF
SENDER
SERVICE
SET
SIGNAL
SIGNALLIST
SIGNALROUTE
SIGNALSET
SPELLING
START
STATE
STOP
keyword4


keyword4:
STRUCT
SUBSTRUCTURE
SYNONYM
SYNTYPE
SYSTEM
TASK
THEN
TYPE
TIMER
TO
VIA
VIEW
VIEWED
WITH
XOR
T1004600-88

ANNEX E

(to Recommendation Z.100)

State-oriented representation and pictorial elements

E.1 *Introduction*

SDL is based on an "extended" Finite State Machine (FSM) model. That is, an FSM is extended with objects, such as variables, resources, etc. A machine stays in some state. On receiving a signal, a machine executes a transition, in which relevant actions (e.g. resource allocation and/or deallocation, resource control, signal sending, decision, etc.) are taken. Therefore, the dynamic behaviour of an extended FSM can be explained by describing action sequences on objects for each transition of the FSM in a procedural way.

As a consequence of the state transition, the machine arrives in a new state. The state of an extended FSM can be characterized by objects associated with the state, additional object information (e.g. the value of variables, states of resources, relations between the resources), and signals which can be received in that state. For example, the "await-first-digit state" in telephone call processing is characterized as follows:

Caller:	handset-off
Dial tone-sender:	dialtone sending
Digit receiver:	ready for receiving
Timer:	supervising permanent-signal timing
Path:	Caller is connected to dial tone-sender and digit receiver, etc.

As can be seen, each state can be defined statically by objects and additional information (qualifying text) associated with that state.

The SDL/GR is extended with **pictorial elements** to define objects associated with each state. The state definitions in terms of pictorial elements are called **state pictures**. The SDL/GR state symbol may include a state picture. This is an optional part of SDL/GR. Figure E-1 shows a state definition example of the "await-first-digit state".

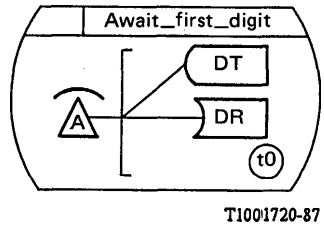


FIGURE E-1

State definition example in terms of pictorial elements

In many cases, actions on each object, which are required in the transition, can be derived from the difference between state definitions before and after the transition. For example, if some resource appears only after transition, it means that resource allocation action is necessary in the transition. Therefore, if detailed state definitions are given, total actions in the extended FSM transition can usually be derived from the difference between pre-and post-state definitions. However, the sequence of actions in the transition may not be derived from the state definition difference. Therefore, in SDL diagrams, when the sequence of actions is less important, those transition actions which can be derived from the state definition need not be described explicitly. Otherwise, it is desirable to describe action sequences explicitly.

An SDL diagram, in which transitions are described exclusively by explicit action symbols, is called a **transition-oriented version of SDL/GR**.

An SDL/GR diagram, in which states are described using state pictures and transition actions are minimized, is called the **state-oriented version of SDL/GR** or **state-oriented SDL with pictorial elements (SDL/PE).** State pictures can be used advantageously when applied to certain system definitions, resulting in more compact, declarative and less verbal process diagrams.

A combined version is also possible. Thus, these are 3 SDL/GR versions:

a) Transition-oriented version
   - Transition sequences are described exclusively by explicit action symbols.
   - This is, as it were, a procedural explanation of the extended FSM.
   - This version is suitable when the sequence of actions is important and detailed state descriptions are not important.

b) State-oriented version
   - The state is described uniquely using pictorial elements. This picture is called a state picture.
   - The transition action sequence is implied by the difference between pre-and post-state definitions.
   - This is, as it were, a declarative specification of the extended FSM.
   - This vesion is suitable when the sequence of actions within each transition is of low importance, when pictorial explanation is desirable, or when a compact representation is desirable.

c) Combined version
   - The combined version is suitable when both the sequence of actions within each transition and the detailed state descriptions are under consideration.

Examples of these three versions are given in Figure E-2, E-3 and E-4.

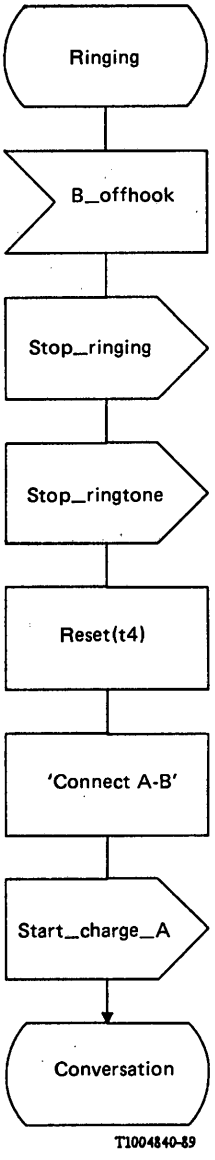


FIGURE E-2

Transition-oriented version

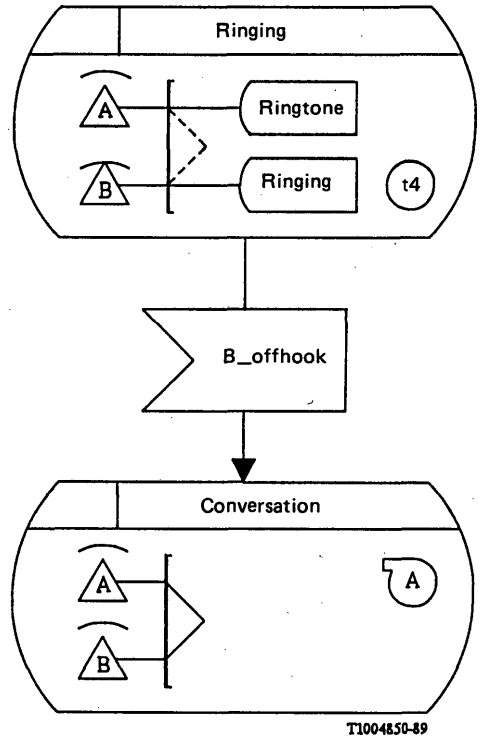


FIGURE E-3

State-oriented version

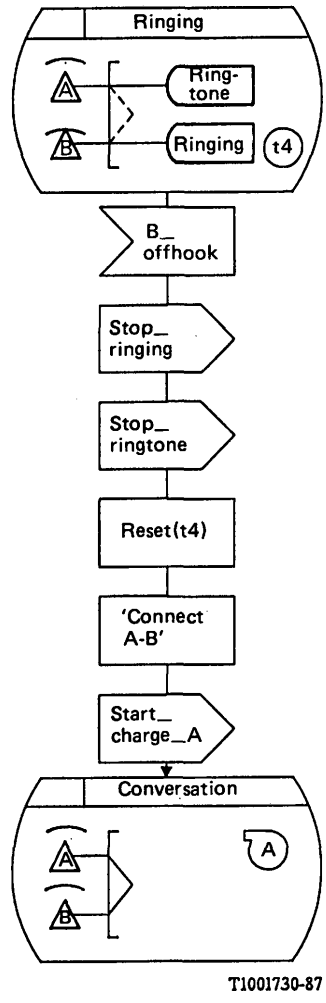


FIGURE E-4

Combined version

E.2 *Pictorial elements in SDL/GR*

The syntax and semantics defined in Recommendation Z.100 SDL applies to pictorial elements. However, these semantics and syntax are extended as follows:

Pictorial elements represent various objects. The repertoire of pictorial elements is in principle unlimited because new pictorial elements can be invented to suit any new application of the SDL. However, in applications to telecommunications switching and signalling functions, the following repertoire of pictorial elements has been found to have considerable versatility:

- functional block boundary (left or right),
- terminal equipment (various),
- signalling receiver,
- signalling sender,
- combined signalling sender and receiver,
- supervising timer,
- switching path (connected, reserved),
- switching modules,
- charging in progress,
- control elements,
- uncertainty symbol.

Standard symbols for these pictorial elements are recommended in section E.2.2.

E.2.1 *Rules of interpretation*

1) A state symbol may include a **state picture**. A state picture defines de state using pictorial elements and qualifying text.

2) Each pictorial element in a state picture represents an object associated with the state, such as:
   - resources,
   - variables,
   - internal and external boundaries,
   - the relations between objects,
   - signals which can be received in that state,
   - etc.

3) Each pictorial element may have accompanying **qualifying text**. Qualifying text can be used to explain:
   - detailed resource name,
   - the resource state,
   - value for a variable,
   - signals relevant to the object,
   - etc.

4) Function block boundary:
   a) A **function block boundary** is used to express whether a pictorial element is "internal" or "external" to the process. An internal pictorial element represents objects which are owned by the process. An external pictorial element represents objects which are owned by another process under consideration.
   b) Rule a) also applies to the distinction between internal and external qualifying text, by substituting the term "qualifying text" for pictorial elements in the rule.

5) Transition interpretation rule:

   The total processing involved when a process goes from one state to the following state is the combination of:
   - The processing to effect changes to all relevant objects which are derived from the state definition difference.
   - The processing explicitly described in the transition, e.g. outputs or tasks.

Thus:

a) The absence from one state if a pictorial element which represents a resource with its presence in the next state implies the allocation of the resource in all transitions joining the two states. This can be equivalently represented by a task(s) showing allocation of the resource in transition(s).

b) If "presence" and "absence" are interchanged in rule a), then "allocation" is replaced by "deallocation".

c) In rule a) if "pictorial element" is replaced by "external pictorial element" then the task should be replaced by an output signal requesting the process which owns the resource to allocate it or simply an input signal from that process saying that it has been allocated.

d) If in rule a) "presence" and "absence" are interchanged and also "pictorial element" is replaced by "external pictorial element" then follow rule c) with "allocate" replaced with "deallocate".

e) Rules a), b), c) and d) also apply to the appearance or disappearance in the state picture of qualifying text, by substituting the term "qualifying text" for pictorial elements in those rules.

6) For a given process diagram, particular pictorial elements (or a particular combination of pictorial elements and qualifying text) should always be placed in the same position within the state picture whenever they appear, so that the presence or absence of this pictorial element (or combination) in a state symbol can be quickly determined by comparing the state picture with other state pictures in the process diagram.

7) When a signal sender appears in a state picture, its qualifying text identifies a signal which is sent during the following transitions.

8) When a sender of a permanent signal (e.g. a ringtone) appears in a state picture, its qualifying text identifies a signal which has been started to be sent during the following transition and in this state.

9) Such transition actions that cannot be derived from the difference of pre- and post-state definitions should be explicitly described in the transition. For example, if a resource with an exported variable does not appear in the pre- and post-states, the necessary actions are better to be described in the transition.

E.2.2 *Recommended symbols for pictorial elements*

When using pictorial elements, each state is represented by a state symbol containing a state picture with the format shown in Figure E-5:

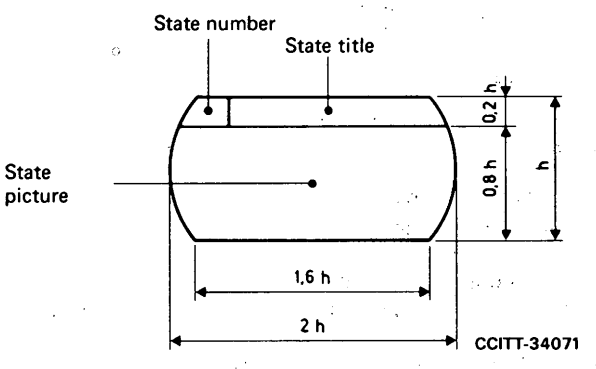


FIGURE E-5

Recommended format for a state symbol with a state picture

A basic set of pictorial elements is recommended for use in SDL/GR with application to the system description of telecommunications call handling processes, including signalling protocols, network services and signalling interworking processes. Many of these pictorial elements are capable of being applied in applications of SDL/GR to other than call handling processes.

The recommended symbols for the basic set of pictorial elements is shown in Figure E-6, and the recommended proprotions for pictorial element symbols are shown in Figure E-7.

Examples of the use of the basic set of pictorial elements are shown in Figure E-8.

E.2.3 *Special conventions and interpretations used in the state oriented extension of SDL/GR*

A number of special conventions and interpretations have been defined in this section with regard to the state-oriented version of SDL/GR. These includes:

- The special interpretation required for process diagrams according to the so-called TRANSITION INTERPRETATION RULE (see § E.2.1, rule 5).
- The unique position of pictorial elements (or pictorial elements and qualifying text) within a state picture that is required when using pictorial elements (see § E.2.1, rule 6).
- The special interpretation required for the variables represented by external pictorial elements and external qualifying text, as proxy variables associated with other processes.

E.3 *Selection criteria for pictorial elements*

The choice of symbols for pictorial elements has been based upon the following considerations and general selection criteria. These should be consulted before developing additional pictorial element symbols for wider applications of the SDL.

1) Ease of reproduction

   In order to permit convenient reproduction of SDL diagrams using the dye-line or blue-print methods of reproduction as well as photocopying and photo-printing, pictorial element symbols should consist of clear lines without shading or coloration.

2) Ease of comprehension

   a) Appropriateness – The shape of each symbol should be appropriate to the concept that the symbol represents.

   b) Distinctiveness – When choosing a basic set of symbols, care should be taken to permit each symbol to be readily distinguishable from others in the set.

   c) Affinity – The shapes of pictorial elements representing different but related functions, e.g. receivers and senders, should be related in some obvious way.

   d) Association of abbreviated qualifying text with symbols – In some cases it is expected that abbreviated text will be associated with a pictorial element in order to indicate the class of pictorial element; e.g. the letters MFC associated with a receiver symbol to indicate that multi-frequency coded signals are to be received. In these cases, the pictorial elements should incorporate enclosed space to permit the use of a very small number of alphanumerical characters.

   e) Limited set – The total number of symbols should be kept to a minimum in order to permit easy learning of the pictorial method.

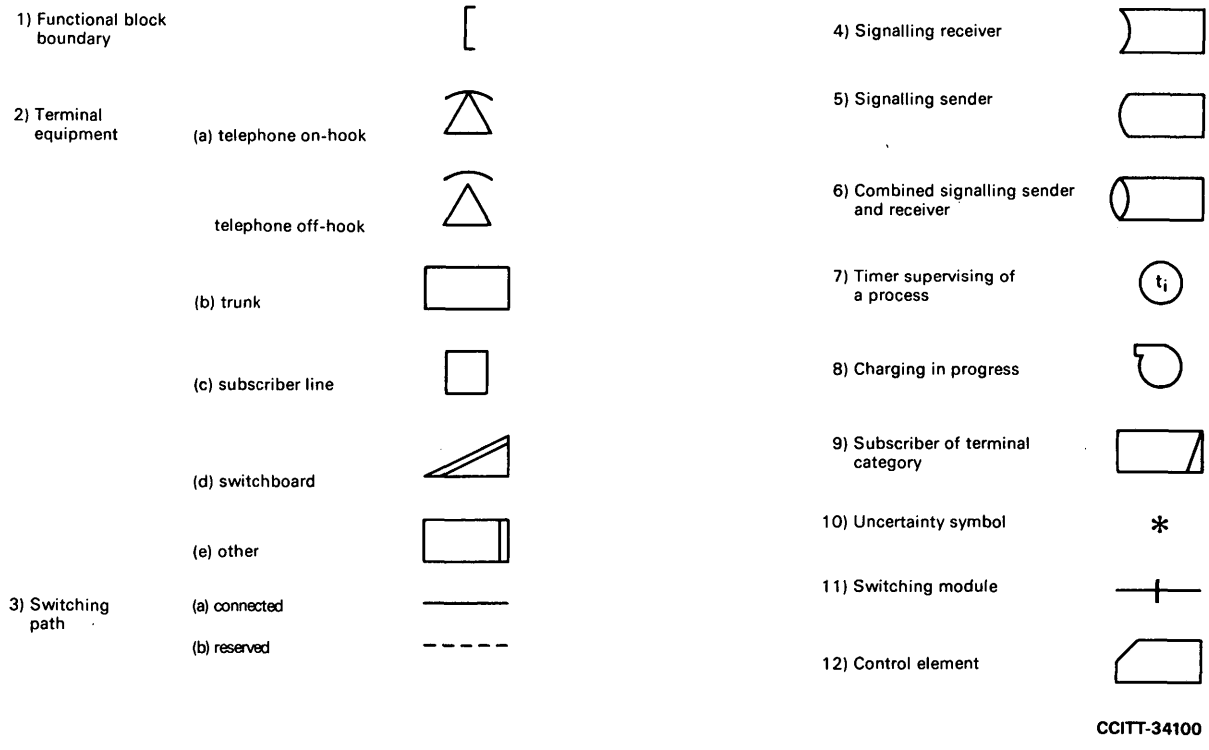


FIGURE E-6

Recommended symbols for the basic set of pictorial concepts

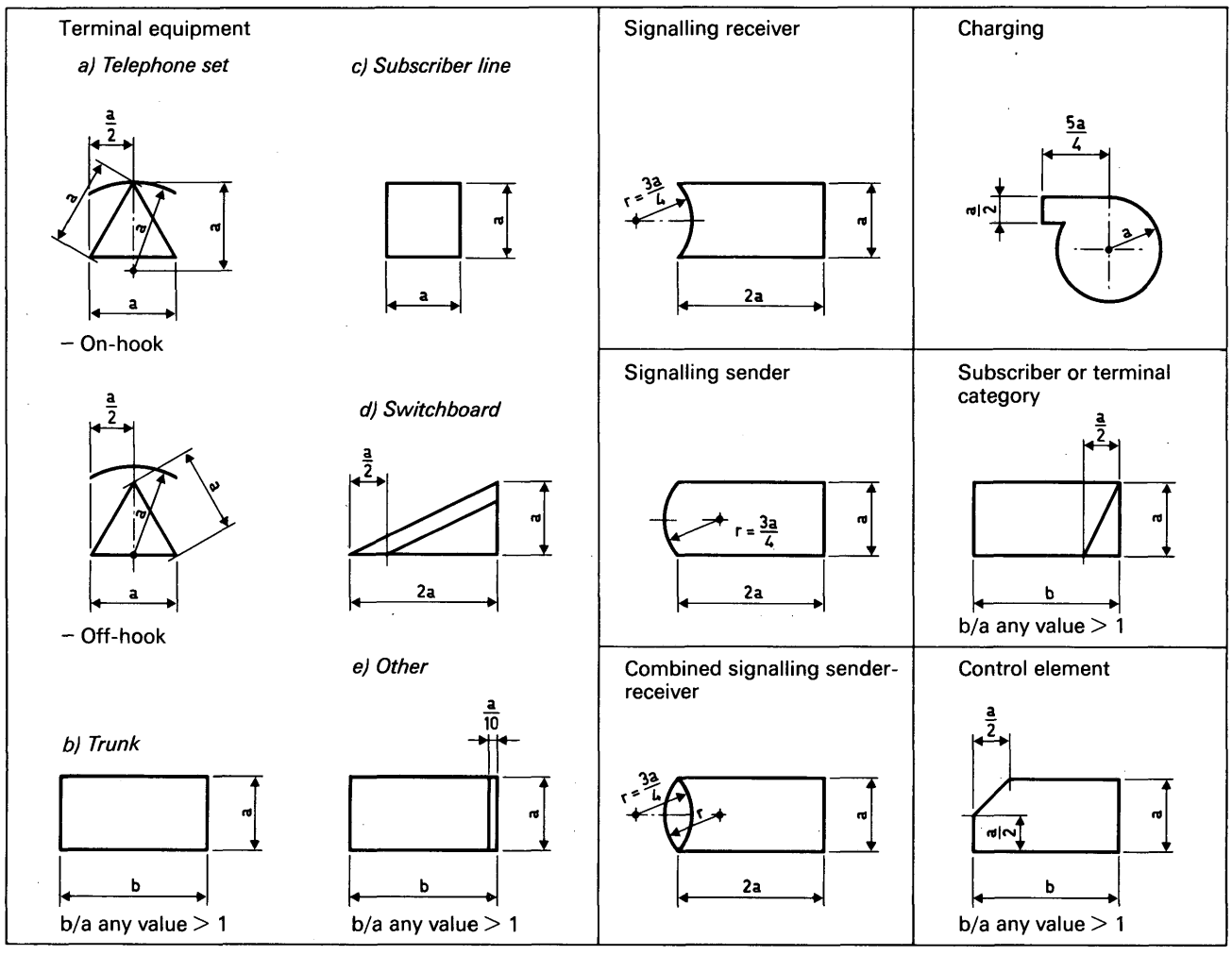
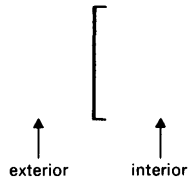
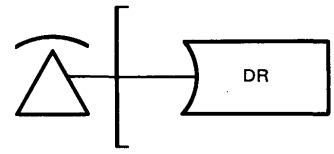
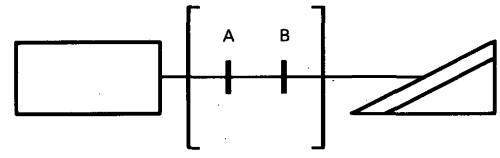
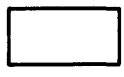
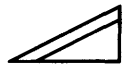
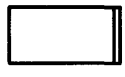
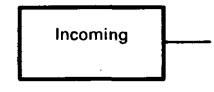
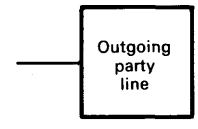
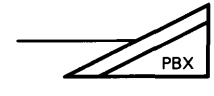
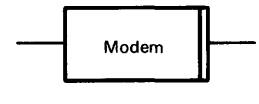


FIGURE E-7

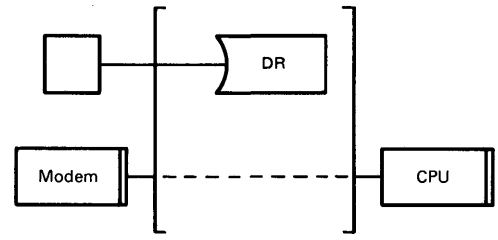
Recommended proportions for the basic sets of pictorial elements

No.	Pictorial element	Comment	Examples
1.	*Functional block* (FB) *boundary* 	To distinguish between objects inside and outside the FB boundary. Only the states of objects within the boundary can be changed directly by this process.	1.1 A digit receiver inside the FB boundary connected to a handset outside the boundary.  1.2 A trunk outside the FB boundary connected via a two-stage switching unit to a switchboard outside the boundary 
2.	*Terminal equipment* a) Telephone set on-hook  off-hook  b) Trunk  c) Subscriber line [except a)]  d) Switchboard  e) Other 	It can be useful to show terminal equipment (e.g. telephone set and switchboard equipment) outside the FB boundary, to improve understanding of the processing work.	2.1 A on-hook  2.2 B off-hook  2.3 Incoming trunk junctor (from a space division switching exchange)  2.4 Outgoing subscriber line to party line  2.5 PBX switchboard  2.6 Modem 

CCITT-20880

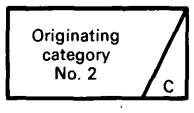
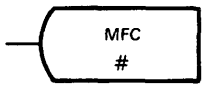
FIGURE E-8

Examples of the use of the basic set of pictorial elements

No.	Pictorial element	Comment	Examples
3.	*Switching path* a) connected ——— b) reserved - - - - - -	To show connectivity between terminal equipment and/or signalling devices involved in the process.	3.1 Subscriber line connected to a digit receiver and a modem with a reserved path to a central processing unit (CPU) 
4.	*Signalling receiver* 	To indicate the nature of the signals received, especially those crossing the functional block boundary.	4.1 Multi-frequency code signalling receiver 
5.	*Signalling sender* 	To specify a signal sending process, and to indicate the nature of the signals sent, especially those required to cross the functional block boundary.	5.1 Ringtone sender 
6.	*Combined signalling sender and receiver* 	This conveniently combines the functions of a signalling sender and signalling receiver.	6.1 MFC sender-receiver 
7.	*Process supervising timer* 	This shows the timer to be running in the state.	7.1 Timer t_3 is running  7.2 Generic timer t_s is running  where s = 1, 2, ... n define different service tones.

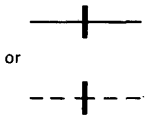
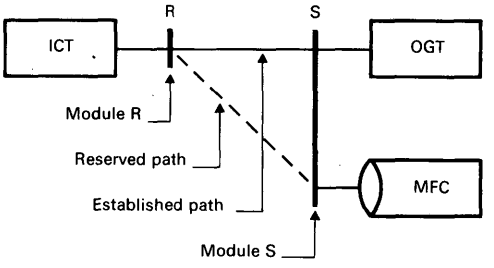
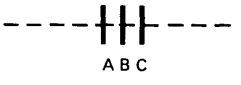
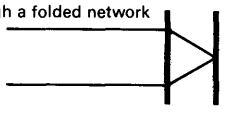
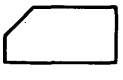
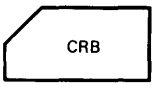
CCITT-20890

FIGURE E-8 (*cont.*)

No.	Pictorial element	Comment	Examples
8.	Charging in progress 	The qualifying text in the element indicates which customer is being charged.	8.1 Subscriber A is currently being charged 
9.	*Subscriber or terminal category (and identity information)* 	This element is convenient to show the changes in the subscriber or terminal category, for each party in a multi-party call.	9.1 The C party has originating category No. 2 
10.	*Uncertainty symbol* 	This substitutes for deliberately undefined information that is shown unambiguously in other state pictures. In certain cases, two or more states may be safely merged into one, with a net gain in the intelligibility of the diagram, by using the uncertainty symbol.	10.1 Handset either on-hook or off-hook  10.2 An undefined MFC signal is being sent in this state 

CCITT-20900

FIGURE E-8 (*cont.*)

No.	Pictorial element	Comment	Examples
11.	*Switching module*  or	To show what switching modules are involved in the process. *Note* – The horizontal line is the pictorial element for a switching path, which may be connected or reserved. The vertical line can be used to represent either a complete switching module (when the internal structure of the module is not required) or else one of the switching stages within a switching module.	11.1 A path connected through one switching module LLN (Line link network) LLN = Line link network  11.2 Paths connected and reserved through two switching modules  ICT – Incoming trunk OGT – Outgoing trunk MFC – Multi-frequency code *Note* – In this example, ICT is connected to OGT, but ICT is not connected to the MFC sender/receiver. 11.3 A path connected through a three-stage switching module RSN  11.4 A path reserved through a three-stage switching module ABC  11.5 A path connected through a folded network 
12.	*Control element (assigned to a process)* 	To show what control equipment is involved in the process (especially modules that must be dimensioned). This symbol can be used to indicate that particular software elements have been assigned to the process.	12.1 Call register buffer 

CCITT-20910

FIGURE E-8 (*end*)

CRITERIA FOR THE USE AND APPLICABILITY OF FORMAL DESCRIPTION TECHNIQUES[1)]

1 Support for formal description techniques (FDTs)

In view of the complexity and widespread use of Recommendations it is imperative that advanced methods for the development and implementation of these Recommendations be used.

Formal description techniques provide an important approach toward such advanced methods.

In some areas, the use of FDTs is still relatively new and phased procedures are required to introduce their use. This Recommendation proposes the procedures to accomplish this task.

2 FDTs

2.1 *Definitions*

A **formal description technique (FDT)** is a specification method based on a description language using rigorous and unambiguous rules both with respect to developing expressions in the language (formal syntax) and interpreting the meaning of these expressions (formal semantics). FDTs are intended to be used in the development, specification, implementation and verification of Recommendations (or parts thereof).

A **natural language description** is an example of an informal description technique using one of the languages used by CCITT to publish Recommendations. It may be supplemented with mathematical and other accepted notation, figures, etc.

2.2 *Objectives of an FDT*

The goal of an FDT is to permit precise and unambiguous specifications. FDTs are also intended to satisfy objectives such as:

- a basis for analyzing specifications for correctness, efficiency, etc.;
- a basis for determining completeness of specifications;
- a basis for verification of specifications against the requirement of the Recommendation;
- a basis for determining conformance of implementations to Recommendations;
- a basis for determining consistency of specifications between Recommendations;
- a basis for implementation support.

In the current state of the art, in some areas more than one FDT may be needed to accomplish all objectives.

2.3 *Benefits of an FDT*

The application of an FDT can provide benefits such as:

- improving the quality of Recommendations, which in turn reduces maintenance costs to both CCITT and to users of Recommendations;
- reducing dependency on the natural language to communicate technical concepts in a multilingual environment;
- reducing development time of implementations by using tools that are based on the properties of the FDT;
- making the implementation easier, resulting in better products.

[1)] The content of this Recommendation is also published as ISO Resolution ISO/IEC JTC 1/N 145.
The statement on precedence in case of several descriptions contained in the JTC 1 document is omitted in this Recommendation.

2.4 *Problem with FDTs*

FDTs are advanced techniques which have not yet been widely introduced. In addition, there is a lack of resources in the development of FDTs and formal descriptions (FDs), as well as a lack of expertise within the CCITT Study Groups both to assess the technical merits of the formally described Recommendations and to reach consensus on them.

2.5 *Solutions*

The development of tutorial and educational materials will help to provide widespread understanding of the complexities of FDTs. Nevertheless, time must be permitted for their assimilation.

3 Development and standardization of FDTs

It is important to avoid unnecessary proliferation of FDTs. The following criteria should be met before adopting a new FDTs:

- the need for the FDT should be demonstrated;
- evidence that it is based on a significantly different model from that of an existing FDT should be provided, and
- the usefulness and capabilities of the FDT should be demonstrated.

4 Development and acceptance of formal descriptions

4.1 In future, only standard FDTs or FDTs in the process of being standardized should be used in formal descriptions of Recommendations.

4.2 It is considered that the development of a FD of any particular Recommendation is a decision of the Study Group (in consultation with ISO for collaborative standards). If a FD is to be developed for a new Recommendation, the FD should be progressed, as far as possible, according to the same timetable as the rest of the Recommendation.

4.3 For the evolutionary introduction of FDs into Recommendations three phases can be identified. It is the responsibility of the Study Group to decide which phase initially applies to each FD and the possible evolution of the FD toward another phase. It is not mandatory for a FD to go through the three phases described below and, more generally, it is not mandatory for a FD to evolve.

Phase 1

This phase is characterized by the fact that widespread knowledge of FDTs, and experience in formal descriptions, are lacking; there may not be sufficient resources in the Study Groups to produce or review formal descriptions.

The development of Recommendations has to be based on conventional natural language approaches, leading to Recommendations where the natural language description is the definitive Recommendation.

Study Groups are encouraged to develop FDs of their Recommendations since these efforts may contribute to the quality of the Recommendations by detecting defects, may provide additional understanding to readers, and will support the evolutionary introduction of FDTs.

A formal description produced by a Study Group that can be considered to represent faithfully a significant part of the Recommendation or the complete Recommendation should be published as an appendix to the Recommendation.

Meanwhile Study Groups should develop and provide educational material for the FDTs to support their widespread introduction in the CCITT and Liaison Organizations.

Phase 2

This phase is characterized by the fact that knowledge of FDTs and experience in formal descriptions is more widely available; Study Groups can provide enough resources to support the production of formal descriptions. However, it cannot be assured that enough CCITT Members can review formal descriptions in order to enable them to approve a proposed formally described Recommendation.

The development of Recommendations should still be based on conventional natural language approaches, leading to Recommendation where the natural language description is the definitive standard. However, these developments should be accompanied and supported by the development of formal descriptions of these standards with the objective of improving and supporting the structure, consistency, and correctness of the natural language description.

A formal description, produced by Study Group, that is considered to represent faithfully a significant part of the Recommendation or the complete Recommendation should be published as an annex to the Recommendation.

Meanwhile educational work should continue.

Phase 3

This phase is characterized by the fact that a widespread knowledge of FDTs may be assumed; CCITT Members can provide sufficient resources both to produce and review formal descriptions, and assurance exists that the application of FDTs does not unnecessarily restrict freedom of the implementations.

Study Groups should use FDTs routinely to develop their Recommendations, and the FD(s) become part of the Recommendation together with natural language descriptions.

Whenever a dicrepancy between a natural language description and a formal description, or between two formal descriptions, is detected, the discrepancy should be resolved by changing or improving the natural language description or the FDs without necessarily giving preference to one over the other(s).

4.4 The above procedures for phased development of FDs are intended to aid the progression of FDs within the standards process, not to hinder their progression. However, since there has been little or no actual experience with these procedures, any Study Group having to use them is urged to identify one or more pilot cases and carefully monitor the progress of each within the framework of the procedures. Should procedural problems arise, the Study Group responsible for Formal Description Techniques should be informed and, where possible, recommended procedural modifications should be proposed to alleviate the problems.

ISBN 92-61-03751-8