



INTERNATIONAL TELECOMMUNICATION UNION

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Z.100

(03/93)

PROGRAMMING LANGUAGES

**CCITT SPECIFICATION AND DESCRIPTION
LANGUAGE (SDL)**

ITU-T Recommendation Z.100

(Previously "CCITT Recommendation")

FOREWORD

The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of the International Telecommunication Union. The ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.

The World Telecommunication Standardization Conference (WTSC), which meets every four years, established the topics for study by the ITU-T Study Groups which, in their turn, produce Recommendations on these topics.

ITU-T Recommendation Z.100 was revised by the ITU-T Study Group X (1988-1993) and was approved by the WTSC (Helsinki, March 1-12, 1993).

NOTES

1 As a consequence of a reform process within the International Telecommunication Union (ITU), the CCITT ceased to exist as of 28 February 1993. In its place, the ITU Telecommunication Standardization Sector (ITU-T) was created as of 1 March 1993. Similarly, in this reform process, the CCIR and the IFRB have been replaced by the Radiocommunication Sector.

In order not to delay publication of this Recommendation, no change has been made in the text to references containing the acronyms “CCITT, CCIR or IFRB” or their associated entities such as Plenary Assembly, Secretariat, etc. Future editions of this Recommendation will contain the proper terminology related to the new ITU structure.

2 In this Recommendation, the expression “Administration” is used for conciseness to indicate both a telecommunication administration and a recognized operating agency.

TABLE OF CONTENTS

	<i>Page</i>
1 Introduction to SDL.....	1
1.1 Introduction.....	1
1.1.1 Objective.....	1
1.1.2 Application.....	1
1.1.3 System specification.....	2
1.2 SDL grammars.....	2
1.3 Basic definitions	3
1.3.1 Definition, type and instance.....	3
1.3.2 Environment.....	5
1.3.3 Errors	5
1.4 Presentation style.....	5
1.4.1 Division of text.....	5
1.4.2 Titled enumeration items.....	5
1.5 Metalanguages	7
1.5.1 Meta IV	7
1.5.2 BNF.....	9
1.5.3 Metalanguage for graphical grammar.....	10
1.6 Differences to SDL-88.....	11
2 Basic SDL	13
2.1 Introduction.....	13
2.2 General rules.....	13
2.2.1 Lexical rules	13
2.2.2 Visibility rules, names and identifiers	16
2.2.3 Informal text.....	20
2.2.4 Drawing rules	20
2.2.5 Partitioning of diagrams	20
2.2.6 Comment	21
2.2.7 Text extension	22
2.2.8 Text symbol.....	22
2.3 Basic data concepts.....	22
2.3.1 Data type definitions	23
2.3.2 Variable	23
2.3.3 Values and literals.	23
2.3.4 Expressions	23
2.4 System structure.....	23
2.4.1 Organisation of SDL specifications.....	23
2.4.1.1 Framework	23
2.4.1.2 Package	24
2.4.1.3 Referenced definition	26
2.4.2 System.....	28
2.4.3 Block.....	30
2.4.4 Process	32
2.4.5 Service.....	37
2.4.6 Procedure	39
2.5 Communication.....	42
2.5.1 Channel	42
2.5.2 Signal route	44

2.5.3	Connection	47
2.5.4	Signal	48
2.5.5	Signal list definition	49
2.6	Behaviour.....	50
2.6.1	Variables	50
2.6.1.1	Variable definition	50
2.6.1.2	View definition.....	51
2.6.2	Start.....	51
2.6.3	State.....	52
2.6.4	Input	53
2.6.5	Save.....	55
2.6.6	Spontaneous transition	56
2.6.7	Label	57
2.6.8	Transition	58
2.6.8.1	Transition body	58
2.6.8.2	Transition terminator.....	59
2.6.8.2.1	Nextstate.....	59
2.6.8.2.2	Join.....	60
2.6.8.2.3	Stop	60
2.6.8.2.4	Return.....	61
2.7	Action	62
2.7.1	Task.....	62
2.7.2	Create	63
2.7.3	Procedure call.....	64
2.7.4	Output	65
2.7.5	Decision	68
2.8	Timer	70
2.9	Internal input and output	71
2.10	Examples.....	72
3	Structural Decomposition Concepts in SDL.....	82
3.1	Introduction.....	82
3.2	Partitioning	82
3.2.1	General.....	82
3.2.2	Block partitioning.....	83
3.2.3	Channel partitioning.....	86
3.3	Refinement.....	89
4	Additional Concepts of Basic SDL.....	91
4.1	Introduction.....	91
4.2	Macro.....	91
4.2.1	Lexical rules	91
4.2.2	Macro definition.....	91
4.2.3	Macro call	95
4.3	Generic system definition	97
4.3.1	External synonym.....	97
4.3.2	Simple expression	97
4.3.3	Optional definition	98
4.3.4	Optional transition string.....	100
4.4	Asterisk state.....	102

4.5	Multiple appearance of state	103
4.6	Asterisk input	103
4.7	Asterisk save	103
4.8	Implicit transition.....	103
4.9	Dash nextstate	104
4.10	Priority Input.....	104
4.11	Continuous signal.....	105
4.12	Enabling condition.....	106
4.13	Imported and Exported value.....	109
4.14	Remote procedures	112
5	Data in SDL.....	115
5.1	Introduction.....	115
5.1.1	Abstraction in data types	115
5.1.2	Outline of formalisms used to model data	115
5.1.3	Terminology	116
5.1.4	Division of text on data	116
5.2	The data kernel language	116
5.2.1	Data type definitions	116
5.2.2	Literals and parameterised operators.....	119
5.2.3	Axioms	121
5.2.4	Conditional equations.....	124
5.3	Passive use of SDL data.....	125
5.3.1	Extended data definition constructs.....	125
5.3.1.1	Special operators.....	126
5.3.1.2	Character string literals	127
5.3.1.3	Predefined data.....	128
5.3.1.4	Equality and noequality	129
5.3.1.5	Boolean axioms	129
5.3.1.6	Conditional terms	130
5.3.1.7	Errors	131
5.3.1.8	Ordering	132
5.3.1.9	Syntypes	132
5.3.1.9.1	Range condition.....	134
5.3.1.10	Structure sorts	136
5.3.1.11	Inheritance.....	137
5.3.1.12	Generators	139
5.3.1.12.1	Generator definition	139
5.3.1.12.2	Generator transformation	141
5.3.1.13	Synonyms	142
5.3.1.14	Name class literals.....	143
5.3.1.15	Literal mapping	144
5.3.2	Operator definitions	146
5.3.3	Use of data	148
5.3.3.1	Expressions	148
5.3.3.2	Ground expressions	149
5.3.3.3	Synonym.....	150
5.3.3.4	Indexed primary	151
5.3.3.5	Field primary	151

	5.3.3.6	Structure primary.....	152
	5.3.3.7	Conditional ground expression.....	153
5.4		Use of data with variables.....	153
	5.4.1	Variable and data definitions.....	153
	5.4.2	Accessing variables.....	154
	5.4.2.1	Active expressions.....	154
	5.4.2.2	Variable access.....	155
	5.4.2.3	Conditional expression.....	155
	5.4.2.4	Operator application.....	156
	5.4.3	Assignment statement.....	157
	5.4.3.1	Indexed variable.....	158
	5.4.3.2	Field variable.....	158
	5.4.3.3	Default initialization.....	159
	5.4.4	Imperative operators.....	160
	5.4.4.1	Now expression.....	160
	5.4.4.2	Import expression.....	161
	5.4.4.3	PId expression.....	161
	5.4.4.4	View expression.....	162
	5.4.4.5	Timer active expression.....	162
	5.4.4.6	Anyvalue expression.....	163
	5.4.5	Value returning procedure call.....	163
	5.4.6	External data.....	164
6		Structural Typing Concepts in SDL.....	166
6.1		Types, instances, and gates.....	166
	6.1.1	Type definitions.....	166
	6.1.1.1	System type.....	166
	6.1.1.2	Block type.....	167
	6.1.1.3	Process type.....	168
	6.1.1.4	Service type.....	170
	6.1.2	Type expression.....	171
	6.1.3	Definitions based on types.....	172
	6.1.3.1	System definition based on system type.....	172
	6.1.3.2	Block definition based on block type.....	173
	6.1.3.3	Process definition based on process type.....	173
	6.1.3.4	Service definition based on service type.....	174
	6.1.4	Gate.....	175
6.2		Context parameter.....	177
	6.2.1	Process context parameter.....	179
	6.2.2	Procedure context parameter.....	179
	6.2.3	Remote procedure context parameter.....	180
	6.2.4	Signal context parameter.....	180
	6.2.5	Variable context parameter.....	181
	6.2.6	Remote variable context parameter.....	181
	6.2.7	Timer context parameter.....	181
	6.2.8	Synonym context parameter.....	181
	6.2.9	Sort context parameter.....	182
6.3		Specialization.....	182
	6.3.1	Adding properties.....	182
	6.3.2	Virtual type.....	183
	6.3.3	Virtual transition/save.....	184
6.4		Examples.....	185
7		Transformation of SDL Shorthands.....	194
	7.1	Transformation of additional concepts.....	194
	7.2	Insertion of full qualifiers.....	198
		Annex A – Index to articles 2 to 7 of Recommendation Z.100 (normative parts).....	200
		Annex B – SDL Glossary.....	220

SUMMARY

Scope-objective

This Recommendation defines SDL (CCITT *Specification and Description Language*) intended for unambiguous specification and description of telecommunications systems. The scope of SDL is elaborated in 1.1.1. This Recommendation is a reference manual for the language.

Coverage

SDL has concepts for behaviour and data description as well as for structuring large systems. The basis of behaviour description is extended finite state machines communicating by messages. The basis for data description is algebraic data types. The basis for structuring is hierarchical decomposition and type hierarchies. These foundations of SDL are elaborated in the respective main sections of the Recommendation. A distinctive feature of SDL is the graphical representation.

Applications

SDL is applicable within standard bodies and industry. The main applications areas, which SDL has been designed for are stated in 1.1.2, but SDL is generally suitable for describing reactive systems.

Status/Stability

This Recommendation is the complete language reference manual supported by guidelines for its usage in Appendix I. Annex F gives a formal definition of SDL semantics.

Associated work

The main text is accompanied by annexes:

- A Index of non-terminals and keywords
- B Glossary
- C Initial Algebra Model
- D SDL Predefined Data
- E reserved for future use
- F Formal Definition

And appendices:

- I SDL Methodology Guidelines
- II SDL Bibliography

One method for SDL usage within standards is described in Recommendation Q.65. A recommended strategy for introducing a formal description technique like SDL in standards is available in Recommendation Z.110. References to additional material on SDL, including information on industrial usage of SDL, can be found in Appendix III.

Background

Different versions of SDL have been recommended by CCITT since 1976. This version is a revision of Z.100, ITU 1988.

Compared to SDL as defined in 1988, the version defined herein has been extended in the area of object-oriented structuring to cope with object-oriented system modelling. Other minor extensions have been included, whereas care has been taken not to invalidate existing SDL-88 documents. Details on the changes introduced can be found in 1.6

Keywords

Abstract data types, formal description technique, functional specification, graphical presentation, hierarchical decomposition, object orientation, specification technique, state machine.

CCITT SPECIFICATION AND DESCRIPTION LANGUAGE (SDL)

(Melbourne, 1988; revised in Helsinki, 1993)

1 Introduction to SDL

The text of clause 1 is not normative; rather it defines the conventions used for describing SDL. The usage of SDL in this section is only illustrative. The metalanguages and conventions introduced are solely introduced for the purpose of describing SDL unambiguously.

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.

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 is intended to specify the behavioural aspects of a system; the general parameters describing properties like capacity and weight have to be described using different techniques.

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.

1.1.1 Objective

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 particular methodology.

1.1.2 Application

This Recommendation is the reference manual for SDL. A methodology guidelines document which gives examples of SDL usage is available as Appendix I.

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;
- f) telecommunications services.

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 descriptions (both high level and detailed);
- g) system testing descriptions

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, 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.

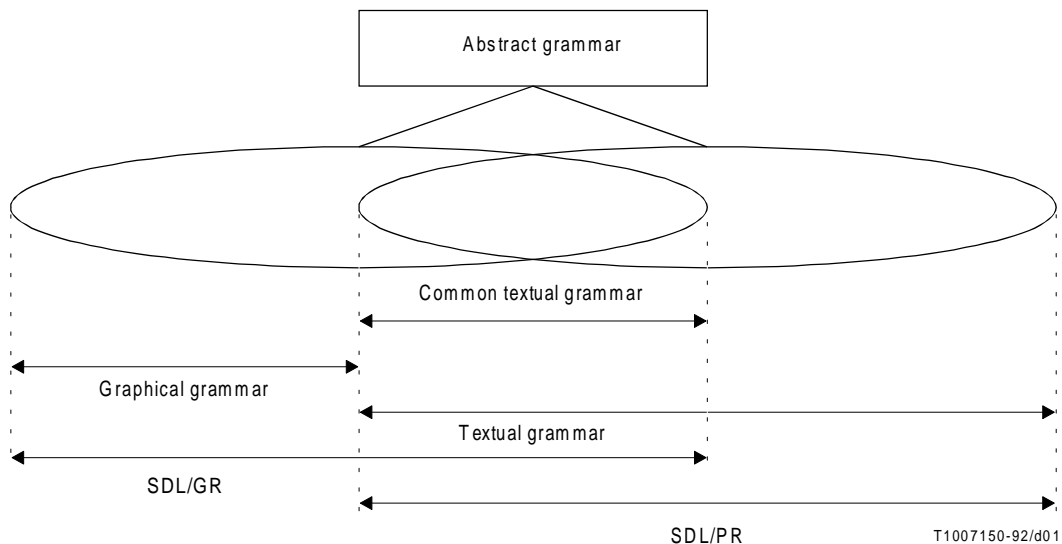


FIGURE 1.1/Z.100
SDL grammars

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 provided which defines how to transform a system specification into the abstract syntax and defines how to interpret a specification, given in terms of the abstract grammar. The formal definition is given in Annex F to this Recommendation.

1.3 Basic definitions

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

1.3.1 Definition, type and instance

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

This section introduces the basic semantics of type definitions, instance definitions, parameterized type definitions, parameterization, binding of context parameters, specialization and instantiation.

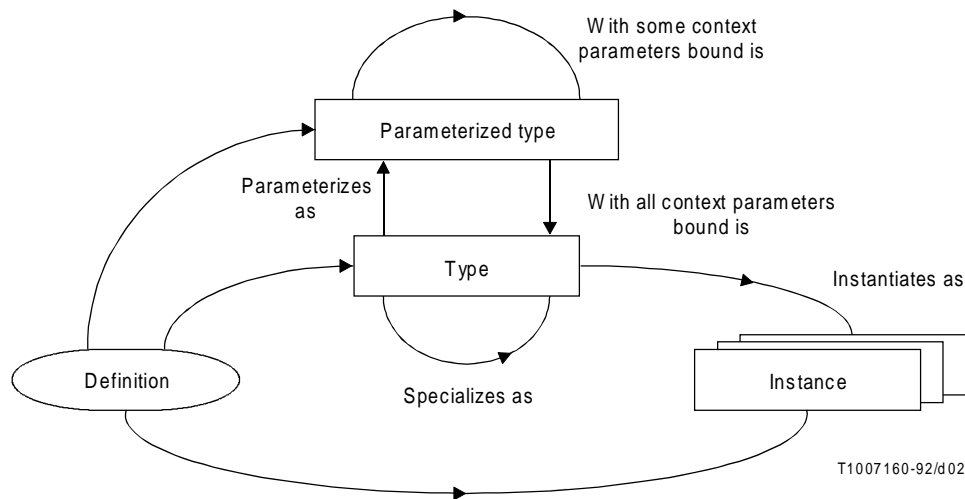


FIGURE 2.2/Z.100
The type concept

Definitions introduce named entities which are either types or instances. A definition of a type defines all properties associated with that type. An example of an instance definition is a variable definition. An example of a definition which is a type definition is a signal definition.

A type may be instantiated in any number of instances. An instance of a particular type has all the properties defined for that type. An example of a type is a procedure, which may be instantiated by procedure calls.

A parameterized type is a type where some entities are represented as formal context parameters. A formal context parameter of a type definition has a constraint. The constraints allow static analysis of the parameterized type. Binding all the parameters of a parameterized type yields an ordinary type. An example of a parameterized type is a parameterized signal definition where one of the sorts conveyed by the signal is specified by a formal sort context parameter; this allows the parameter to be of different sorts in different contexts.

An instance is defined either directly or by the instantiation of a type. An example of an instance is a system instance which can be defined by a system definition or be an instantiation of a system type.

Specialization allows one type, the subtype, to be based on another type, its supertype, by adding properties to those of the supertype or by redefining virtual properties of the supertype. A virtual property may be constrained in order to provide for analysis of general types.

Binding all context parameters of a parameterized type yields an unparameterized type. There is no supertype/subtype relationship between a parameterized type and the unparameterized type derived from it.

Data type is a special kind 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 towards the system have associated identities of these process instances. These processes have PId values which are distinguishable from any of the PId values within the system (see Annex D, D.10).

Although the behaviour of the environment is non-deterministic, it is assumed to 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

The Recommendation is organized 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 construct, provides the properties it has, the way in which it is interpreted and any dynamic conditions which have to be fulfilled for the construct to behave well 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 2.1/Z.200 (see also 1.5.2 of this Recommendation).

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 and relevant static conditions from the concrete textual 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 referenced definitions (2.4.1.3), macros (4.2) and literal mappings (5.3.1.15), etc., must be considered before the correspondence between the concrete and the abstract syntax. These expansions are detailed in clause 7.

Semantics

Properties are relations between different concepts in SDL. Properties are used in the well-formedness rules.

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 clause 2. This is usually not 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 notation) 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 *Example(s)* contains example(s).

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 view definition is

$$\textit{View-definition} \quad :: \quad \begin{array}{l} \textit{Variable-identifier} \\ \textit{Sort-reference-identifier} \end{array}$$

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

The Meta IV definition

$$\textit{Process-identifier} \quad = \quad \textit{Identifier}$$

expresses that a *Process-identifier* is an *Identifier* and therefore cannot 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.

$$\textit{Number-of-instances} \quad :: \quad \textit{Intg} \quad [\textit{Intg}]$$

Number-of-instances denotes a composite domain containing one mandatory integer (*Intg*) value and one optional integer ([*Intg*]) denoting respectively the initial number and the optional maximum number of instances.

- b) Quotation objects

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

Example.

$$\textit{Destination} \quad = \quad \textit{Process-identifier} \mid \textit{Service-identifier} \mid \mathbf{ENVIRONMENT}$$

The *Destination* is either a *Process-identifier*, *Service-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 to consist of a potentially infinite set of distinct atomic objects for which no representation is required.

Example.

$$\textit{Name} \quad :: \quad \textit{Token}$$

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.

$$\textit{Informal-text} \quad :: \quad \dots$$

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 angle 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 angle 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 angle brackets. For each non-terminal symbol, a production rule is given either in concrete textual grammar or in graphical grammar. For example,

<textual block reference> ::=
 block <block name> **referenced** <end>

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. For example, <textual block reference>, <block name> and <end> in the example above are non-terminals; **block** and **referenced** are terminal symbols.

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

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> ::=
 <block diagram>
 | <existing typebased block definition>
 | <graphical block reference>
 | <graphical typebased block definition>

expresses that a <block area> is either a <graphical block reference>, a <block diagram>, a <graphical typebased block definition> or an <existing typebased block definition>.

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 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,

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.

1.6 Differences to SDL-88

The language defined in this Recommendation is an extension of Z.100 as published in the 1988 *Blue Books*. In this section, the language defined in the *Blue Books* will be called SDL-88 and the language defined in this Recommendation will be called SDL-92. Every effort has been made to make SDL-92 a pure extension of SDL-88, without invalidating the syntax or changing the semantics of any existing SDL-88 usage. In addition, enhancements were only accepted based on need as supported by several CCITT member-bodies.

The major extensions are in the area of object orientation. While SDL-88 is object based in its underlying model, some language constructs have been added to allow SDL-92 to more completely and uniformly support the object paradigm (see 2.4.1.2 and 6):

- a) packages (2.4.1.2);
- b) system, block, process and service types (6.1.1);
- c) system, block, process and service (set of) instances based on types (6.1.2);
- d) parameterization of types by means of context parameters (6.2);
- e) specialization of types, and redefinition of virtual types and transitions (6.3).

The other extensions are: spontaneous transition (2.6.6), non-deterministic choice (2.7.5), internal input and output symbol in SDL/GR for compatibility with existing diagrams (2.9), a non-deterministic imperative operator **any** (5.4.4.6), non-delaying channel (2.5.1), remote procedure call (4.14) and value returning procedure (5.4.5), input of variable field (2.6.4), operator definition (5.3.2), combination with external data descriptions (5.4.6), extended addressing capabilities in output (2.7.4), free action in transition (2.6.7), continuous transitions in same state with same priority (4.11), m:n connections of channels and signal routes at structure boundaries (2.5.3). In addition, a number of minor relaxations to the syntax have been introduced.

In a few cases, it has been necessary to make changes to SDL-88. These have been introduced solely where the definition of SDL-88 is not consistent. The restrictions and changes introduced can be overcome by an automatic translation procedure. This procedure is also necessary, if an SDL-88 document contains names consisting of words which are keywords of SDL-92.

For the **output** construct the semantics have been simplified, and this may invalidate some special usage of **output** (when no **to** clause is given and there exist several possible paths for the signal) in SDL-88 specifications. Also, some properties of the equality property of sorts have been changed.

For the **export/import** construct an optional remote variable definition has been introduced, in order to align export of variables with the introduced export of procedures (remote procedure). This necessitates a change to SDL-88 documents, which contain qualifiers in import expressions or introduce several imported names in the same scope with different sorts. In the (rare) cases where it is necessary to qualify import variables to resolve resolution by context, the correction is to introduce <remote variable definition>s and to qualify with the identifier of the introduced remote variable name.

For the **view** construct, the view definition has been made local to the viewing process or service. This necessitates a change to SDL-88 documents, which contain qualifiers in view definitions or in view expressions. The correction is to remove these qualifiers. This will not change the semantics of the view expressions, since these are decided by their (unchanged) Pid-expressions.

The **service** construct has been defined as a primitive concept, instead of being a shorthand, without extending its properties. The use of service is not affected by this change, since it has been used anyway as if it were a primitive concept. The reason for the change is to simplify the language definition and align it with the actual use, and to reduce the number of restrictions on service, caused by the transformation rules in SDL-88. As a consequence of this change the service signal route construct has been deleted, signal routes can be used instead. This is only a minor conceptual

change, and has no implications for concrete use (the syntax of SDL-88 service signal route and SDL-92 signal route are the same).

The **priority output** construct has been removed from the language. This construct can be replaced by **output to self** with an automatic translation procedure.

Some of the definitions of basic SDL may appear to have been extended considerably, e.g. **signal** definition. But it should be noted that the extensions are optional and need only be used for utilising the power introduced by the object oriented extensions, e.g. to use parameterization and specialization for signals.

Keywords of SDL-92 which are not keywords of SDL-88 are:

any, as, atleast, connection, endconnection, endoperator, endpackage, finalized, gate, interface, nodelay, noequality, none, package, redefined, remote, returns, this, use, virtual.

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.

Clause 2 is divided into nine main topics:

a) *General rules*

Basic 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 data concepts such as values, variables, expressions.

c) *System structure*

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

d) *Communication*

Contains communication mechanisms 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, spontaneous transition, label, transition.

f) *Action*

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

g) *Timer*

Timer definition and timer primitives.

h) *Internal Input and Output*

Shorthands for compatibility with older versions of SDL.

i) *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> ::=	<word> <character string> <special> <composite special> <note> <keyword>
<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>

```

<apostrophe> <apostrophe> represents an <apostrophe> within a <character string>.

```

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

```

```

<apostrophe> ::= '

```

```

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

```

```

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

```

```

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

```

```

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

```

```

<keyword> ::=
    active          | adding          | all
    alternative      | and            | any
    as               | at least       | axioms
    block            | call           | channel
    comment          | connect        | connection
    constant         | constants      | create
    dcl              | decision       | default
    else             | endalternative | endblock
    endchannel       | endconnection  | enddecision
    endgenerator     | endmacro       | endnewtype
    endoperator      | endpackage     | endprocedure
    endprocess       | endrefinement  | endselect
    endservice       | endstate       | endsubstructure
    endsyntype       | endsystem      | env
    error            | export         | exported
    external         | fi             | finalized
    for              | fpar           | from
    gate             | generator      | if
    import           | imported       | in
    inherits         | input          | interface
    join             | literal        | literals
    macro            | macrodefinition| macroid
    map              | mod            | nameclass
    newtype          | nextstate      | nodelay
    noequality       | none           | not
    now              | offspring      | operator

```

	operators		or		ordering
	out		output		package
	parent		priority		procedure
	process		provided		redefined
	referenced		refinement		rem
	remote		reset		return
	returns		revealed		reverse
	save		select		self
	sender		service		set
	signal		signallist		signalroute
	signalset		spelling		start
	state		stop		struct
	substructure		synonym		syntype
	system		task		then
	this		timer		to
	type		use		via
	view		viewed		virtual
	with		xor		

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.

Control characters are defined as in Recommendation T.50. A sequence of control characters may appear where a <space> may appear, and has the same meaning as a <space>. The <space> represents the CCITT Alphabet No. 5 character for a space.

An occurrence of a control character is not significant in <informal text> and in <note>. A control character cannot appear in character string literals if its presence is significant. In these cases the operator // and the literals for control characters must be used.

In all <lexical unit>s except <character string>, <letter>s are always treated as if uppercase. (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 <space>s, 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 split over more than one line.

When the character / is immediately followed by the character * outside of a <note>, it 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>.

2.2.2 Visibility rules, names and identifiers

Abstract grammar

<i>Identifier</i>	::	<i>Qualifier Name</i>
<i>Qualifier</i>	=	<i>Path-item</i> +
<i>Path-item</i>	=	<i>System-qualifier</i> <i>Block-qualifier</i> <i>Block-substructure-qualifier</i> <i>Signal-qualifier</i> <i>Process-qualifier</i> <i>Service-qualifier</i> <i>Procedure-qualifier</i> <i>Sort-qualifier</i>

<i>System-qualifier</i>	::	<i>System-name</i>
<i>Block-qualifier</i>	::	<i>Block-name</i>
<i>Block-substructure-qualifier</i>	::	<i>Block-substructure-name</i>
<i>Process-qualifier</i>	::	<i>Process-name</i>
<i>Service-qualifier</i>	::	<i>Service-name</i>
<i>Procedure-qualifier</i>	::	<i>Procedure-name</i>
<i>Signal-qualifier</i>	::	<i>Signal-name</i>
<i>Sort-qualifier</i>	::	<i>Sort-name</i>
<i>Name</i>	::	<i>Token</i>

Concrete textual grammar

<code><name> ::=</code>	<code><word> {<underline> <word>}*</code>
<code><identifier> ::=</code>	<code>[<qualifier>] <name></code>
<code><qualifier> ::=</code>	<code><path item> {/<path item>}*</code> <code><< <path item> {/<path item>}* >></code>
<code><path item> ::=</code>	<code><scope unit kind> {<name> <quoted operator>}</code>
<code><scope unit kind> ::=</code>	<code>package</code> <code>system type</code> <code>system</code> <code>block</code> <code>block type</code> <code>substructure</code> <code>process</code> <code>process type</code> <code>service</code> <code>service type</code> <code>procedure</code> <code>signal</code> <code>operator</code> <code>type</code>

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`. This rule does not apply to the use of `<underline>` and `<space>` in `<character string>`.

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 transformations>` then the `<underline>`s in the `<sort name>` following the keyword **newtype** must be specified explicitly.
- 4) The `<underline>`s in `<process name>` of a `<process context parameter>` with `<process constraint>` being `<process identifier>` only, must be specified explicitly.

- 5) The <underline>s in the <sort> in <procedure result> must be specified explicitly.

<quoted operator> is only applicable when <scope unit kind> is operator, see 5.3.2.

There is no corresponding abstract syntax for the <scope unit kind> denoted by package, system type, block type, process type, service type or operator.

Either the <qualifier> refers to a supertype or the <qualifier> reflects the hierarchical structure from the system or package level to the defining context, such that the system or package level is the leftmost textual part.

It is allowed to omit some of the leftmost <path item>s, or the whole <qualifier>. When the <name> denotes an entity of the entity kind 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 does not apply.

The qualification by supertype makes visible names of virtual types in a supertype otherwise hidden by a redefinition in the subtype (see 6.3.2). This is, the names in the supertype which can be qualified by supertype.

If a <qualifier> can be understood both as qualifying by an enclosing scope and as qualifying by a supertype, it denotes an enclosing scope.

A subsignal must be qualified by its parent signal if other visible signals (without context parameters) exist at that place with 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 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 compatibility, 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 <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:

<i>Concrete textual grammar</i>	<i>Concrete graphical grammar</i>
<package definition>	<package diagram>
<textual system definition>	<system diagram>
<system type definition>	<system type diagram>
<block definition>	<block diagram>
<block type definition>	<block type diagram>
<process definition>	<process diagram>
<process type definition>	<process type diagram>
<service definition>	<service diagram>
<service type definition>	<service type diagram>
<procedure definition>	<procedure diagram>
<block substructure definition>	<block substructure diagram>
<channel substructure definition>	<channel substructure diagram>
<partial type definition>	
<operator definition>	<operator diagram>
<sort context parameter>	
<signal definition>	
<signal context parameter>	

A scope unit has a list of definitions attached. Each of the definitions defines an entity belonging to a certain entity kind and having an associated name. Included in the list are <gate definition>s, <formal context parameter>s, <formal parameters>s, <formal variable parameters> and substructure definitions contained in the scope unit. 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 <specialization> or <ordering>.

Although <quoted operator>s, <operator name>s with an <exclamation> and <character string>s have their own syntactical notation, they are in fact <name>s that are in the *Abstract syntax* represented by a *Name*. In the following, they are treated as if they were syntactically also <name>s. However, <state name>s, <connector name>s, <gate name>s occurring in signal route and channel definitions, <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 single <body>. 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 kinds exist:

- a) packages
- b) system
- c) system types
- d) block types
- e) blocks
- f) channels, signal routes, gates
- g) block substructures, channel substructures
- h) signals, timers
- i) process types
- j) processes
- k) service types
- l) services
- m) procedures, remote procedures
- n) variables (including formal parameters), synonyms, literals, operators
- o) remote variables
- p) sorts
- q) generators
- r) signal lists
- s) view

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 and the <identifier> either
 - 1) does not occur in a <gate definition>, an <actual context parameter>, a <parameters of sort>, a <formal variable parameters> or a <specialization>, or
 - 2) denotes a <formal context parameter>; or
- b) if it is visible in the scope unit which defines 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; or
- e) if the scope unit has a <package reference clause> through which the <identifier> is made visible (as defined in 2.4.1.2).

No two definitions in the same scope unit and belonging to the same entity kind can have the same <name>. The only exception is operators and literal definitions in the same <partial type definition>. These can have the same <name> with different <argument sort>s or different <result> sort.

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 provide 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 area>. 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>, <internal input symbol>, <internal output symbol>, <priority input symbol>, <comment symbol> and <text extension symbol> are allowed.

The right-hand argument of the metasymbol **is associated with** must be closer to the left-hand argument than to any other graphical symbol. The syntactical elements of the right-hand 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 right-hand 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>
<heading> ::=	<kernel heading> [<additional heading>]
<kernel heading> ::=	[<virtuality>] [exported] <diagram kind> {< <u>diagram</u> name> < <u>diagram</u> identifier> }
<page number area> ::=	<implicit text symbol> contains [<page number> [(<number of pages>)]]
<page number> ::=	< <u>literal</u> name>
<number of pages> ::=	<Natural literal name>

$$\langle \text{heading} \rangle ::= \langle \text{kernel heading} \rangle [\langle \text{additional heading} \rangle]$$

```

<kernel heading> ::=
    [<virtuality>] [exported]
    <diagram kind> {<diagram name> | <diagram identifier>}

```

<page number area> ::=

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

`<page number> ::=`
`<literal name>`

$$\langle \text{number of pages} \rangle ::= \langle \text{Natural literal name} \rangle$$

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 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>` depend on the type of diagram.

<additional heading> must be shown on the first page of a diagram, but is optional on the following pages. <heading> and <diagram kind> are elaborated for the particular diagrams in the individual sections of this Recommendation. <additional heading> is not defined further by this Recommendation.

<virtuality> denotes the virtuality of the type defined by the diagram (see 6.3.2) and **exported** whether a procedure is exported as a remote procedure (see 4.14).

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>.

Examples are shown in Figures 2.10.1 to 2.10.4.

The concrete syntax of the second form is:

```
<end> ::=
```

```
<comment> ::=
    comment <character string>
```

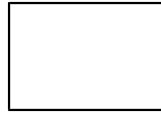
An example is shown in Figure 2.10.2.

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

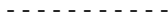
<comment area> ::=

<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.

<text> in *Concrete graphical grammar* corresponds to <character string> in *Concrete textual grammar* without the enclosing <apostrophe>s.

An example is shown in Figure 2.10.4.

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 clause 5. This includes the data terminology, the concepts to define new sorts and the predefined data.

2.3.1 Data type definitions

Data in SDL is principally concerned with sorts. A sort 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 sort. These properties are defined by sort definitions.

SDL allows the definition of any needed sort, 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 sorts 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 values of the Boolean sort is not.

Each value belongs to exactly 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 (e.g. 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 or an imperative operator, e.g. 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 or imperative operators may be interpreted as having different values during the interpretation of an SDL system depending on the value associated with the variables.

2.4 System structure

2.4.1 Organisation of SDL specifications

2.4.1.1 Framework

An <sdl specification> can be described as a monolithic <system definition> or as a collection of <package>s and <referenced definition>s. A <package> allows definitions to be used in different contexts by “using” the package in these contexts, i.e. in systems or packages which may be independent. A <referenced definition> is a definition that has been removed from its defining context to gain overview within one system description. It is similar to a macro definition (see 4.2), but it is “inserted” into exactly one place (the defining context) using a reference.

Concrete grammar

<sdl specification> ::=

<package list> [<system definition> {<referenced definition>}*]

2.4.1.2 Package

In order for a type definition to be used in different systems it has to be defined as part of a *package*.

Definitions as parts of a package define types, data generators, signal lists, remote specifications and synonyms.

Definitions within a package are made visible to a system or other packages by a *package reference clause*.

Concrete textual grammar

```
<package list> ::=
    { <package> {<referenced definition>}* }*

<package> ::=
    <package definition> | <package diagram>

<package definition> ::=
    {<package reference clause>}*
    package <package name>
        [<interface>] <end>
        {<entity in package>}*
    endpackage [<package name>] <end>

<entity in package> ::=
    <system type definition>
    | <textual system type reference>
    | <block type definition>
    | <textual block type reference>
    | <process type definition>
    | <textual process type reference>
    | <procedure definition>
    | <remote procedure definition>
    | <textual procedure reference>
    | <signal definition>
    | <signal list definition>
    | <service type definition>
    | <textual service type reference>
    | <select definition>
    | <remote variable definition>
    | <data definition>
    | <macro definition>

<package reference clause> ::=
    use <package name> [ / <definition selection list> ] <end>

<definition selection list> ::=
    <definition selection> { , <definition selection> }*

<definition selection> ::=
    [<entity kind>] <name>

<entity kind> ::=
    system type
    | block type
    | process type
    | service type
    | signal
    | procedure
    | newtype
    | signallist
```


	generator
	synonym
	remote

<interface> ::=

interface <definition selection list>

newtype is also used for selection of a syntype name in a package. The <entity kind> **remote** is used for selection of a remote variable definition.

Concrete graphical grammar

<package diagram> ::=

<package reference area>
is associated with
 <frame symbol> **contains**
 { <package heading>
 { {<package text area>}*
 {<diagram in package> }* } **set** }

<package reference area> ::=

<text symbol> **contains** {<package reference clause>}*

<package heading> ::=

package <package name> [<interface>]

<package text area> ::=

<text symbol> **contains**
 { <entity in package> }*

<diagram in package> ::=

	<system type diagram>
	<system type reference>
	<type in system area>
	<macro diagram>
	<option area>

The <package reference area> must be placed on the top of the <frame symbol>.

Semantics

For each <package name> mentioned in a <package reference clause>, there must exist a corresponding <package>. If this <package> is not part of <package list>, there must exist a mechanism for accessing the referenced <package>, just as if it were a textual part of the <package list>. This mechanism is not defined in this Recommendation.

Likewise, if the <system definition> is omitted in an <sdl specification>, there must exist a mechanism for using the <package>s in the <package list> in other <sdl specification>s. Before the <package list> is used in other <sdl specification>s, the model for macros and referenced definitions is applied. The mechanism is not otherwise defined in this Recommendation.

The name of an entity defined within a <package> is visible in another <package> or the <system definition> if and only if:

- 1) the <package> or <system definition> has a <package reference clause> mentioning the <package> and the <package reference clause> either has the <definition selection list> omitted or the name is mentioned in a <definition selection>; and
- 2) the <package> defining the name either has the <interface> omitted or the name is mentioned in the <interface>.

Resolution by context in expressions will take into account also those sorts in packages which are not made visible in a <package reference clause>. Signals which are not made visible in a **use** clause, can be part of a signal list via a <signal list identifier> made visible in a **use** clause and these signals can thereby affect the complete valid input signal set of a process or service.

If a name in a <definition selection> denotes a <sort>, the <definition selection> also implicitly denotes all the literals and operators defined for the <sort> or <parent sort identifier> in case of a syntype.

If a name in a <definition selection> denotes a signal, the <definition selection> also implicitly denotes all the subsignals of that signal.

Names having their defining occurrence in a <package> are referred to by means of <identifier>s having **package** <package name> as the leftmost <path item>. However, the <path item> may be omitted if either the <package name> denotes the enclosing package, if (name, entity kind) is visible only from one package or if resolution by context applies.

The <entity kind> in <definition selection> denotes the entity kind of the <name>. Any pair of (<entity kind>, <name>) must be distinct within a <definition selection list>. For a <definition selection> in an <interface>, the <entity kind> may be omitted only if there is no other name having its defining occurrence directly in the <package>. For a <definition selection> in a <package reference clause>, <entity kind> may be omitted if and only if either exactly one entity of that name is mentioned in any <definition selection list> for the package or the package has no <definition selection list> and directly contains a unique definition of that name.

A <system definition> and every <package definition> has an implicit <package reference clause>:

use Predefined;

where Predefined denotes a package containing the predefined data as defined in Annex D.

Model

If a package is mentioned in several <package reference clause>s of a <package definition>, this corresponds to one <package reference clause> which selects the union of the definitions selected in the <package reference clause>s.

An <sdl specification> with <system definition> and a non-empty <package list> corresponds to a <system definition>, where

- 1) all occurrences of the same name (including the defining occurrence) of any entity defined within a <package> have been renamed to the same, unique anonymous name;
- 2) all definitions within the <package>s have been included on the system level;
- 3) all occurrences of **package** <package name> in qualifiers have been replaced by **system** <system name>, where <system name> is the name for <system definition>.

The transformation is detailed in clause 7. The relation of <system definition> to *Abstract Grammar* is defined in 2.4.2.

2.4.1.3 Referenced definition

Concrete grammar

<referenced definition> ::=

<definition> | <diagram>

<system definition> ::=

{<textual system definition> | <system diagram>}

```

<definition> ::=
    <system type definition>
  | <block definition>
  | <block type definition>
  | <process definition>
  | <process type definition>
  | <service definition>
  | <service type definition>
  | <procedure definition>
  | <block substructure definition>
  | <channel substructure definition>
  | <macro definition>
  | <operator definition>

```

```

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

```

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

An <identifier> is present in a <referenced definition> after the initial keyword(s). For each reference there must exist a <referenced definition> with the same entity kind as the reference, and whose <qualifier>, if present, denotes a path, from a scope unit enclosing the reference, to the reference. If two <referenced definition>s of the same entity kind have the same name, then the <qualifier> of one of the <identifier>s must not constitute the leftmost part of the other <qualifier>.

It is not allowed to specify a <qualifier> in the <identifier> after the initial keyword(s) for definitions which are not <referenced definition>s (i.e. a <name> must be specified for normal definitions).

Semantics

Before the properties of a <system definition> are derived, each reference is replaced by the corresponding <referenced definition>. In this replacement, the <identifier> of the <referenced definition> is replaced by the <name> in the reference.

Model

The relation of <referenced definition> to *Abstract grammar* is given in clause 7.

2.4.2 System

Abstract grammar

System-definition :: *System-name*
Block-definition-set
Channel-definition-set
Signal-definition-set
Data-type-definition
Syntype-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 signals, channels, sorts and syntypes, used in the interface with the environment and between blocks of the system are contained in the *System-definition*.

Concrete textual grammar

<textual system definition> ::=

```
{ <package reference clause> } *
{
    system <system name> <end>
    { <entity in system> } +
    endsystem [ <system name> ] <end>
|
    <textual typebased system definition> }
```

<entity in system> ::=

```
<block definition>
|
<textual block reference>
|
<textual typebased block definition>
|
<channel definition>
|
<signal definition>
|
<signal list definition>
|
<select definition>
|
<macro definition>
|
<remote variable definition>
|
<data definition>
|
<textual block type reference>
|
<block type definition>
|
<process type definition>
|
<textual process type reference>
|
<procedure definition>
|
<textual procedure reference>
|
<remote procedure definition>
|
<service type definition>
|
<textual service type reference>
```

<textual block reference> ::=

```
block <block name> referenced <end>
```

An example of <textual system definition> is shown in Figure 2.10.5.

Concrete graphical grammar

<system diagram> ::=

```

    [<package reference area>]
    is associated with
    { <frame symbol> contains
      { <system heading>
        { { <system text area> } *
          { <macro diagram> } *
          <block interaction area>
          { <type in system area> } * } set }
      | <graphical typebased system definition> }
  
```

<frame symbol> ::=



<system heading> ::=

system <system name>

<system text area> ::=

```

<text symbol> contains
  { <signal definition>
    | <signal list definition>
    | <remote variable definition>
    | <data definition>
    | <remote procedure definition>
    | <macro definition>
    | <select definition> } *
  
```

<block interaction area> ::=

{ <block area> | <channel definition area> } +

<block area> ::=

```

    <graphical block reference>
  | <block diagram>
  | <graphical typebased block definition>
  | <existing typebased block definition>
  
```

<graphical block reference> ::=

<block symbol> **contains** <block name>

<block symbol> ::=



The <package reference area> must be placed on the top of the system frame symbol.

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

An example of a <system diagram> is shown in Figure 2.10.6.

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

2.4.3 Block

Abstract grammar

<i>Block-definition</i>	::	<i>Block-name</i> <i>Process-definition-set</i> <i>Signal-definition-set</i> <i>Channel-to-route-connection-set</i> <i>Signal-route-definition-set</i> <i>Data-type-definition</i> <i>Syntype-definition-set</i> [<i>Block-substructure-definition</i>]
<i>Block-name</i>	=	<i>Name</i>

Unless a *Block-definition* contains a *Block-substructure-definition*, there must be at least one *Process-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>
    { <channel to route connection> | <entity in block> } *
    [
        <block substructure definition>
        | <textual block substructure reference> ]
    endblock [ <block name> | <block identifier> ] <end>
```

```
<entity in block> ::=
    <signal definition>
    | <signal list definition>
    | <process definition>
    | <textual process reference>
    | <textual typebased process definition>
    | <signal route definition>
    | <macro definition>
    | <remote variable definition>
    | <data definition>
    | <select definition>
    | <process type definition>
    | <textual process type reference>
```

	<block type definition>
	<textual block type reference>
	<textual procedure reference>
	<procedure definition>
	<remote procedure definition>
	<service type definition>
	<textual service type reference>

<textual process reference> ::=

process <process name> [<number of process instances>] **referenced** <end>

An example of <block definition> is shown in Figure 2.10.7.

Concrete graphical grammar

<block diagram> ::=

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

The <channel identifiers> identify channels connected to signal routes in the <block diagram>. Channel identifier(s) are 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 typebased process definition>
	<existing typebased process definition>

<graphical process reference> ::=

<process symbol> **contains**
 {<process name> [<number of process instances>]}

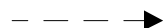
<process symbol> ::=



<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> of a process upon which a create action is performed. <create line symbol>s are optional, but if used then there must in the process at the originating end of the <create line symbol> be a create request for the process at the arrowhead end of the <create line symbol>. This rule applies after transformation of <option area>.

NOTE – This rule can be independently applied before or after transformation of <transition option>.

If a <block definition> or a <block type definition>, which is used in a <textual typebased block definition>, contains <signal route definition>s and <textual typebased process definition>s, then each gate of the <process type definition>s of the <textual typebased process definition>s must be connected to a signal route.

The same rule applies for <process definition>s and <process type definition>s containing <signal route definition>s and <textual typebased service definition>s.

An example of <block diagram> is shown in Figure 2.10.8.

Semantics

A block definition is a container for one or more process definitions of a system and/or a block substructure.

A block 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 process instances in the block.

2.4.4 Process

A process definition defines a arbitrarily, large set of process instances.

Abstract grammar

<i>Process-definition</i>	::	<i>Process-name</i> <i>Number-of-instances</i> <i>Process-formal-parameter</i> * <i>Procedure-definition-set</i> <i>Signal-definition-set</i> <i>Data-type-definition</i> <i>Syntype-definition-set</i> <i>Variable-definition-set</i> <i>View-definition-set</i> <i>Timer-definition-set</i> <i>Process-graph</i> <i>Service-decomposition</i>
<i>Number-of-instances</i>	::	<i>Intg</i> [<i>Intg</i>]
<i>Process-name</i>	=	<i>Name</i>
<i>Process-graph</i>	::	<i>Process-start-node</i> <i>State-node-set</i>
<i>Process-formal-parameter</i>	::	<i>Variable-name</i> <i>Sort-reference-identifier</i>
<i>Service-decomposition</i>	::	<i>Service-definition-set</i> <i>Signal-route-definition-set</i> <i>Signal-route-to-route-connection-set</i>

If *Process-definition* has a *Service-decomposition*, there must be no *Timer-definitions*. *Service-decomposition* must contain at least one *Service-definition*.

<process definition> ::=

```

process { <process name> | <process identifier> }
    [<number of process instances>] <end>
    [<formal parameters> <end>] [<valid input signal set>]
    {
        <entity in process>
        | <signal route to route connection> }*
    [<process body> ]
endprocess [ <process name> | <process identifier> ] <end>

```

<entity in process> ::=

```

    <signal definition>
    | <signal list definition>
    | <textual procedure reference>
    | <procedure definition>
    | <remote procedure definition>
    | <imported procedure specification>
    | <macro definition>
    | <remote variable definition>
    | <data definition>
    | <variable definition>
    | <view definition>
    | <select definition>
    | <imported variable specification>
    | <timer definition>
    | <signal route definition>
    | <service definition>
    | <textual service reference>
    | <textual typebased service definition>
    | <service type definition>
    | <textual service type reference>

```

<textual procedure reference> ::=

```

    <procedure preamble>
    procedure <procedure name> referenced <end>

```

<textual service reference> ::=

```

    service <service name> referenced <end>

```

<valid input signal set> ::=

```

    signalset [<signal list>] <end>

```

<process body> ::=

```

    <start> { <state> | <free action> } *

```

<formal parameters> ::=

```

    fpar <parameters of sort> {, <parameters of sort>}*

```

<parameters of sort> ::=

```

    <variable name> {, <variable name>}* <sort>

```

<number of process instances> ::=

```

    ([<initial number>][,<maximum number>]])

```

<initial number> ::=

```

    <Natural simple expression>

```

<maximum number> ::=

```

    <Natural simple expression>

```

<service definition>, <textual service reference> or <textual typebased service definition> may be present only if <process body> is omitted.

A <process definition> may contain <signal route definition>s only if the enclosing <block definition> or <textual typebased block definition> contains <signal route definition>s.

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

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

- a) If there is no <textual process reference> for the process, then the <number of process instances> in the <process definition> or in the <textual typebased process definition> is used. If it does not contain a <number of process instances>, then the <number of process instances> where both <initial number> and <maximum number> are omitted is used.
- b) If both the <number of process instances> in <process definition> and the <number of process instances> in a <textual process reference> are omitted, then the <number of process instances> where both <initial number> and <maximum number> are omitted is used.
- c) If either the <number of process instances> in <process definition> or the <number of process instances> in a <textual process reference> are omitted, then the <number of process instances> is the one which is present.
- d) If both the <number of process instances> in <process definition> and the <number of process instances> in a <textual process reference> are specified, then the two <number of process instances> must be equal lexically and this <number of process instances> is used.

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

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 2.10.9.

Concrete graphical grammar

<process diagram> ::=

```
<frame symbol>
contains { <process heading>
           { { <process text area> } *
             { <macro diagram> } *
             { <type in process area> } *
             { <process graph area> | <service interaction area> } set }
           [is associated with { <signal route identifiers> } *]
```

The <signal route identifiers> identify external signal routes connected to signal routes in the <process diagram>. It is placed outside the <frame symbol> close to the endpoint of the signal routes at the <frame symbol>.

```

<process text area> ::=
    <text symbol> contains {
        [<valid input signal set>]
        {<signal definition>
        | <signal list definition>
        | <variable definition>
        | <view definition>
        | <imported variable specification>
        | <imported procedure specification>
        | <remote procedure definition>
        | <remote variable definition>
        | <data definition>
        | <macro definition>
        | <timer definition>
        | <select definition> }* }


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

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

<service interaction area> ::=
    {
        <service area>
    | <signal route definition area> }+

<service area> ::=
    <graphical service reference>
    | <service diagram>
    | <graphical typebased service definition>
    | <existing typebased service definition>

<graphical service reference> ::=
    <service symbol> contains <service name>

<service symbol> ::=
    

```

An example of <process diagram> is shown in Figure 2.10.10.

Semantics

The following is applicable when the *Process-definition* contains a *Process-graph*. The additional semantics for the case when the *Process-definition* contains *Service-definitions* are treated in 2.4.5.

A process definition defines a set of processes. Several instances of the same process set may exist at the same time and execute asynchronously and in parallel with each other and with instances of other process sets in the system.

The first value in the *Number-of-instances* represents the number of instances of the process which exist when the system is created, the second value represents the maximum number of simultaneous instances of the process.

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

The concept of finite state machine has been extended in the following respects:

- a) Variables and branching: the state resulting after a transition, other than the signal originating the transition, may be affected by decisions taken upon variables known to the process.
- b) Spontaneous transition: it is possible to activate a transition spontaneously without the reception of any signal. This activation is non-deterministic, and thus makes the state to which it is attached unstable.
- c) Save: with the save construct, the consumption order of signals may be specified to be different from their arrival order in the queue.

When a system is created, the initial processes are created in arbitrary order. The signal communication between the processes commences only when the initial processes have all been created. The formal parameters of these initial processes have no associated values; i.e. they are “undefined”.

Process instances exist from the time that a system is created or they can be created by create request actions of processes being interpreted; their interpretation start when their 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 from 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 set of instances denoted by the process definition, the <valid input signal set>, the implicit input signals introduced by the additional concepts in 4.10 to 4.14 and the timer signals.

Exactly 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. A transition may also be initiated as a spontaneous transition independent of any signals being present in the queue.

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.

The formal parameters are variables which are created either when the system is created (but no actual parameters are passed to them and therefore they are “undefined”) or when the process instance is dynamically created.

To all process instances four expressions yielding a PId (see Annex D, D.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 instance (**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 5.4.4.3.

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 expressions, **sender** and **offspring** have the value Null.

2.4.5 Service

The service concept offers an alternative to the process graph through a set of services. In many situations, service definitions can reduce the overall complexity and increase the readability as compared to the use of a process body. In addition, each service definition may define a partial behaviour of the process, which may be useful in some applications.

Abstract grammar

Service-definition :: *Service-name*
Procedure-definition-set
Data-type-definition
Syntype-definition-set
Variable-definition-set
View-definition-set
Timer-definition-set
Service-graph

Service-name = *Name*

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

Service-start-node :: *Transition*

Concrete textual grammar

<service definition> ::=

```

service { <service name> | <service identifier> } <end>
    [<valid input signal set>]
    {<entity in service>}*
    <service body>
endservice [ { <service name> | <service identifier> } ] <end>

```

<entity in service> ::=

```

    <variable definition>
    | <data definition>
    | <view definition>
    | <imported variable specification>
    | <imported procedure specification>
    | <select definition>
    | <macro definition>
    | <procedure definition>
    | <textual procedure reference>
    | <timer definition>

```

<service body> ::=

```

    <process body>

```

Different <service definition>s or <textual typebased service definition>s in a <process definition> must neither reveal the same variable name of the same sort, export or import the same variable name, nor export or import the same remote procedure.

If an exported variable or procedure is defined in the enclosing process or an imported variable or procedure is specified in the enclosing process, the defined/specified variable/procedure must not be defined/specified with same name as the corresponding <remote variable name> or <remote procedure name> in one of the <service definition>s or <textual typebased service definition>s, and it may only be used as exported/imported in one service. The implicit signals for an

exported variable are added to an arbitrary service if it is not used in any service. An exported procedure defined in the enclosing process must be mentioned as exported in exactly one service.

The complete valid input signal sets of the services within a process must be disjoint. The complete valid input signal set of a service is the union of the <valid input signal set> of its <service definition> and the set of signals conveyed on incoming signal routes to the service including the implicit input signals introduced by the additional concepts in 4.10 to 4.14 and the timer signals.

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>
  |
  <imported variable specification>
  |
  <imported procedure specification>
  |
  <select definition>
  |
  <macro definition> } *

```

<service graph area> ::=

```

<process graph area>

```

Semantics

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 addressed 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 state machine.

When the process instance is created, the *Service-start-nodes* are executed in arbitrary order. No state of any service is interpreted before all *Service-start-nodes* have been completed. A *Service-start-node* is considered completed when the service instance for the first time enters a *State-node* (possibly inside a called procedure) or interprets a *Stop-node*.

Only one service at a time is executing a *Transition*. When the executing service reaches a state, the next signal in the input port (which is not saved by the service, otherwise capable of consuming it) is given to the service that is capable of consuming it.

When a service ceases to exist, the input signals for that service are discarded. When all services have ceased to exist, the process instance ceases to exist.

Example

See 2.10.

2.4.6 Procedure

Procedures are defined by means of procedure definitions. The procedure is invoked by means of a procedure call identifying the procedure definition. Parameters are associated with a procedure call. Which variables are affected by the interpretation of a procedure is controlled by the parameter passing mechanism. Procedure calls may be actions or part of expressions (value returning procedures only).

Abstract grammar

Procedure-definition :: *Procedure-name*
*Procedure-formal-parameter**
Procedure-definition-set
Data-type-definition
Syntype-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 preamble>
    procedure { <procedure name> | <procedure identifier> }
        [<formal context parameters>]
        [<virtuality constraint>]
        [<specialization>] <end>
        [<procedure formal parameters> <end>]
        [<procedure result> <end>]
        {
            <entity in procedure>
            | <select definition>
            | <macro definition> }*
        [<procedure body> ]
    endprocedure [<procedure name> | <procedure identifier>] <end>

```

<procedure preamble> ::=

```

    [<virtuality>][exported [ as <remote procedure identifier> ]]

```

<procedure formal parameters> ::=

```

    fpar <formal variable parameters>
        {, <formal variable parameters> }*

```

<formal variable parameters> ::=

```

    <parameter kind> <parameters of sort>

```

<parameter kind> ::=

[**in/out** | **in**]

<procedure result> ::=

returns [<variable name>] <sort>

<entity in procedure> ::=

<data definition>
| <variable definition>
| <textual procedure reference>
| <procedure definition>

<procedure body> ::=

<start> { <state> | <free action> } *
| { <state> | <free action> } +

An exported procedure cannot have formal context parameters and its enclosing scope must be a process type, process definition, service type or service definition.

<variable definition> in a <procedure definition>, cannot contain **revealed**, **exported**, **revealed exported**, **exported revealed** <variable name>s (see 2.6.1).

The exported attribute is inherited by any subtype of a procedure. A virtual exported procedure must contain **exported** in all redefinitions. Virtual type including procedure is described in 6.3.2. The optional **as**-clause in a redefinition must denote the same <remote procedure identifier> as in the supertype. If omitted in a redefinition, the <remote procedure identifier> of the supertype is implied.

Two exported procedures in a process, including possible services, cannot mention the same <remote procedure identifier>.

An example of <procedure definition> is shown in Figure 2.10.11.

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 preamble>
<procedure name> }

<procedure symbol> ::=



<procedure heading> ::=

<procedure preamble>
procedure { <procedure name> | <procedure identifier> }
[<formal context parameters>]
[<virtuality constraint>]
[<specialization>]
[<procedure formal parameters>]
[<procedure result>]

<procedure graph area> ::=

[<procedure start area>
{ <state area> | <in-connector area> }*]

<procedure start area> ::=

<procedure start symbol>
contains { [<virtuality>] }
is followed by <transition area>

<procedure start symbol> ::=



An example of <procedure diagram> is shown in Figure 2.10.12.

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.

exported in a <procedure preamble> implies that the procedure may be called as a remote procedure, according to the model in 4.14.

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.

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

- A local variable is created for each *In-parameter*, having the *Name* and *Sort* of the *In-parameter*. The variable gets the value of the expression given by the corresponding actual parameter if present. Otherwise the variable gets no value, i.e. it becomes “undefined”.
- A formal parameter with no explicit attribute has an implicit **in** attribute.
- A local variable is created for each *Variable-definition* in the *Procedure-definition*.
- Each *Inout-parameter* denotes a variable which is given in the actual parameter expression. The contained *Variable-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.

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

The nodes of the procedure graph are interpreted in the same manner as the equivalent nodes of a process or service graph, i.e. the procedure has the same complete valid input signal set as the enclosing process, and the same input port as the instance of the enclosing process that has called it, either directly or indirectly.

Model

Specifying <procedure result> results in <procedure formal parameters> with an extra formal variable parameter appended having the **in/out** attribute, a distinct new name (unless specified explicitly in <procedure result>) as <variable name>, and the <sort> of <procedure result> as <sort>.

When a <procedure body> or <procedure graph area> contains <return> or <return area> with an <expression>, the procedure must have at least one formal parameter with <parameter kind> **in/out** or a <procedure result>.

Any occurrence of <return> or <return area> with an <expression> inside the <procedure body> or <procedure graph area> is replaced by a <task> containing an assignment of the contained <expression> to the rightmost **in/out** variable, followed by a <return> or <return area> without any <expression>.

The transformations take place after *Generic systems* (see 4.3) and before <value returning procedure call>.

A <procedure start area> which contains <virtuality> is called a virtual procedure start area. Virtual procedure start area is further described in 6.3.3.

2.5 Communication

2.5.1 Channel

Abstract grammar

<i>Channel-definition</i>	::	<i>Channel-name</i> [NODELAY] <i>Channel-path</i> [<i>Channel-path</i>]
<i>Channel-path</i>	::	<i>Originating-block</i> <i>Destination-block</i> <i>Signal-identifier-set</i>
<i>Originating-block</i>	=	<i>Block-identifier</i> ENVIRONMENT
<i>Destination-block</i>	=	<i>Block-identifier</i> ENVIRONMENT
<i>Block-identifier</i>	=	<i>Identifier</i>
<i>Signal-identifier</i>	=	<i>Identifier</i>
<i>Channel-name</i>	=	<i>Name</i>

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

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

NODELAY denotes that the channel has no delay.

Concrete textual grammar

```

<channel definition> ::=
    channel <channel name> [nodelay]
        <channel path>
        [ <channel path>]
            [ <channel substructure definition>
            | <textual channel substructure reference>]
    endchannel [ <channel name>] <end>

<channel path> ::=
    from <channel endpoint>
    to <channel endpoint> with <signal list> <end>

<channel endpoint> ::=
    { <block identifier> | env } [ via <gate>]

```

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

<gate> must be specified if and only if:

- <channel endpoint> denotes a connection to a <textual typebased block definition> in which case the <gate> must be defined directly in the block type for that block, or
- env is specified and the channel is defined in a block substructure of a <block type definition> in which case the <gate> must be defined in this block type.

Concrete graphical grammar

```

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

```

<channel identifiers> can be specified if and only if the enclosing diagram is a <block substructure diagram> directly enclosed by a <block definition>. <block identifier> can be specified if and only if the enclosing diagram is a <channel substructure diagram>. The <channel identifiers> identify external channels connected to the <block substructure diagram> delimited by the <frame symbol>. The <block identifier> identifies an external block as a channel endpoint for the <channel substructure diagram> delimited by the <frame symbol>. When the <channel symbol> is connected to a <frame symbol>, then the <gate> it is associated with is a gate defined for that block type by means of a <gate> or a <graphical gate constraint> associated with the block type diagram.

```

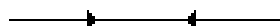
<channel symbol> ::=
    <channel symbol 1>
    | <channel symbol 2>
    | <channel symbol 3>
    | <channel symbol 4>
    | <channel symbol 5>

```

<channel symbol 1> ::=



<channel symbol 2> ::=



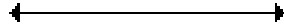
<channel symbol 3> ::=



<channel symbol 4> ::=



<channel symbol 5> ::=



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.

The arrowheads for <channel symbol 4> and <channel symbol 5> are placed at the end(s) of the channel and indicate that the channel has no delay.

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.

The *Signal-identifier-set* in each *Channel-path* in the *Channel-definition* contains the signals that may be conveyed on that *Channel-path*.

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 with delay 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.

If one endpoint of a channel is a <textual typebased block definition>, the other endpoint may be the same <textual typebased block definition>. The <number of block instances> must then be greater than one for the <textual typebased block definition>.

Model

A channel with both endpoints being gates of one <textual typebased block definition> represents transportation routes from each of the blocks in the set to all other blocks in the set.

A channel with one endpoint being a gate of a <textual typebased block definition> represents individual channels to or from each of the blocks in the set. The individual channels will all have the same delaying property as the expanded channel.

2.5.2 Signal route

Abstract grammar

Signal-route-definition ::= *Signal-route-name*
Signal-route-path
[*Signal-route-path*]

<i>Signal-route-path</i>	::	<i>Origin</i> <i>Destination</i> <i>Signal-identifier-set</i>
<i>Origin</i>	=	<i>Process-identifier</i> <i>Service-identifier</i> ENVIRONMENT
<i>Destination</i>	=	<i>Process-identifier</i> <i>Service-identifier</i> ENVIRONMENT
<i>Signal-route-name</i>	=	<i>Name</i>
<i>Process-identifier</i>	=	<i>Identifier</i>
<i>Service-identifier</i>	=	<i>Identifier</i>

At least one of the end points of the signal route must be a *Process-identifier* or *Service-identifier*.

If both end points are processes, the *Process-identifiers* must be different. If both end points are services, the *Service-identifiers* must be different.

The process or service end point(s) must be defined in the same scope unit in which 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 <signal route endpoint> **to** <signal route endpoint>
with <signal list> <end>

<signal route endpoint> ::=

{ <process identifier> | <service identifier> | **env** }
 [**via** <gate>]

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


<gate> must be specified if and only if:


- <signal route endpoint> denotes a connection to a <textual typebased process definition> or a <textual typebased service definition> in which case the <gate> must be defined directly in the process type or service type for that process or service respectively, or
- env** is specified and the signal route is defined in a block type or process type in which case the <gate> must be defined in this block type or process type respectively.

Concrete graphical grammar

<signal route definition area> ::=
 <signal route symbol>
 is associated with { <signal route name>
 { [<channel identifiers> | <external signal route identifiers> |
 <gate>]
 <signal list area> [<signal list area>] } *set* }
 is connected to
 { { <process area> | <service area> }
 { <process area> | <service 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 a <frame symbol>, for a <block type diagram>, then the <gate> it is associated with, is a gate defined for that block type by means of a <gate> or <graphical gate constraint> associated with the <block type diagram>. When the <signal route symbol> is connected to a <frame symbol>, for a <process type diagram>, then the <gate> it is associated with, is a gate defined for that process type by means of a <gate> or <graphical gate constraint> associated with the <process type diagram>.

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 sets of process instances each denoted by a process definition, or between one set of process instances and the environment of the enclosing block, or between two services, or between a service and the environment of the enclosing process.

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

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

When a signal instance is sent to an instance of the same process instance set, 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

If a <block definition> or <block type definition> contains <signal route definition>s, then the <valid input signal set> in a <process definition>, if any, need not contain signals in signal routes leading to the set of process instances.

If a <process type definition> or <process definition> contains services, the <valid input signal set> is derived as the union of the input signals in the gates (in case of a <process type definition>) or the signals in signal routes leading to the process (in case of a <process definition>), the input signals on signal routes leading to the services and the <valid input signal set>s for the services. This set is called implicit <valid input signal set>. If the <valid input signal set> of the process is specified explicitly, it must be a subset of the implicit <valid input signal set>.

If a <process definition> or <process type definition> contains <signal route definition>s, then the <valid input signal set> in a <service definition>, if any, need not contain signals in signal routes leading to the service.

If a <block definition> contains no <signal route definition>s, then all <process definition>s in that scope must (implicitly or explicitly) 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 block boundary.

If a <block type definition> contains no <signal route definition>s, then all <process definition>s in that scope must (implicitly or explicitly) contain a <valid input signal set>. In that case the <signal route definition>s are derived from the <valid input signal set>s, <output>s and the <signal list>s in the gates of the <block type definition>.

The signals corresponding to a given direction between two sets of process instances 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 or gate (in case of a block type) in the given direction.

In case of a process definition being a <textual typebased process definition>, the derivation is not based on the <valid input signal set>s and the <output>s, but on the incoming and outgoing <signal list>s of the gates of the process type.

If a <process definition> or <process type definition> contains no <signal route definition>s, then all <service definition>s in that scope must contain a <valid input signal set>. In that case the <signal route definition>s and the <signal route to route connection>s are derived from the <valid input signal set>s, <output>s and external signal routes terminating at the process boundary or gates of the process type.

The signals corresponding to a given direction between two services in the implied signal route is the intersection of the signals specified in the <valid input signal set> of the destination service and the signals mentioned in an output of the originating service. If one of the endpoints is the environment, then the input set/output set for that endpoint is the signals conveyed by the external signal route or gate (in the case of a process type) in the given direction.

In case of a service definition being a <textual typebased service definition>, the derivation is not based on the <valid input signal set>s and the <output>s, but on the incoming and outgoing <signal list>s of the gates of the service type.

2.5.3 Connection

Abstract grammar

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

Signal-route-identifier = *Identifier*

Signal-route-to-route-connection :: *External-signal-route-identifier-set*
Signal-route-identifier-set

External-signal-route-identifier = *Identifier*

Other connect constructs are contained in clause 3.

The *Channel-identifiers* in a *Channel-identifier-set* in a *Channel-to-route-connection* must denote channels connected to the enclosing block.

Each *Channel-identifier* connected to the enclosing block must be mentioned in exactly one *Channel-to-route-connection*. Each *Signal-route-identifier* in a *Channel-to-route-connection* must be defined in the same block in which 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 exactly 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 union of the *Signal-identifier* sets conveyed by the *Channel-identifiers* in the same *Channel-to-route-connection* and corresponding to the same direction.

The *External-signal-route-identifiers* in an *External-signal-route-identifier-set* in a *Signal-route-to-route-connection* must denote signal routes connected to the enclosing process.

Each *External-signal-route-identifier* connected to the enclosing process must be mentioned in exactly one *Signal-route-to-route-connection*. Each *Signal-route-identifier* in a *Signal-route-to-route-connection* must be defined in the same process in which the *Signal-route-to-route-connection* is defined and it must have the boundary of that process as one of its endpoints. Each *Signal-route-identifier* defined in the surrounding process and which has its environment as one of its endpoints, must be mentioned in exactly one *Signal-route-to-route-connection*.

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

Concrete textual grammar

```

<channel to route connection> ::=
    connect <channel identifiers>
    and <signal route identifiers> <end>

<channel identifiers> ::=
    <channel identifier> {, <channel identifier>}*

<signal route identifiers> ::=
    <signal route identifier> {, <signal route identifier>}*

<signal route to route connection> ::=
    connect <external signal route identifiers>
    and <signal route identifiers> <end>

<external signal route identifiers> ::=
    <signal route identifier>
    {, <signal route identifier>}*

```

Concrete graphical grammar

Graphically, the connect construct is represented by the <channel identifiers> and <external signal route identifiers> associated with the signal routes and contained in the <signal route definition area> (see 2.5.2 *Concrete graphical grammar*).

No <signal route identifier> or <channel identifier> may be mentioned more than once, in the connections of a diagram.

2.5.4 Signal

Abstract grammar

```

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

```

```

Signal-name            =      Name

```


<signal definition> ::=

signal
 <signal definition item>
 {,<signal definition item>}* <end>

<signal definition item> ::=

 <signal name>
 [<formal context parameters>]
 [<specialization>]
 [<sort list>][<signal refinement>]

<sort list> ::=

 (<sort> { , <sort>}*)

<formal context parameter> in <formal context parameters> must be a <sort context parameter>. The <base type> as part of <specialization> must be a <signal identifier>.

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. A signal instance is created when an *Output-node* is interpreted and ceases to exist, when an *Input-node* is interpreted.

2.5.5 Signal list definition

A <signal list identifier> may be used in <signal list> as a shorthand for a list of signal identifiers and timer signals.

Concrete textual grammar

<signal list definition> ::=

 signallist <signal list name> = <signal list> <end>

<signal list> ::=

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

<signal list item> ::=

 <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 list of <signal identifier>s or <timer identifier>s they denote, corresponds to a *Signal-identifier-set* in the *Abstract grammar*.

The <signal list> must not contain the <signal list identifier> defined by the <signal list definition> either directly or indirectly (via another <signal list identifier>).

Concrete graphical grammar

<signal list area> ::=

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

$$\langle \text{signal list symbol} \rangle ::=$$

Model

The *Ground-expression* is represented by:

- a) if a <ground expression> is given in the <variable definition>, then this <ground expression>;
- b) else, if the <sort> has a <default initialization>, then the <ground expression> of the <default initialization>.

Otherwise, the *Ground-expression* is not present.

2.6.1.2 View definition

Abstract grammar

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

In the enclosing *Block-definition* at least one *Process-definition* must exist which contains a **REVEALED** *Variable-definition* with same *Sort-reference-identifier* and same *Variable-name* as the *Name* of *View-name*.

Concrete textual grammar

<view definition> ::=

viewed
<view name> {, <view name>* <sort>
{, <view name> {, <view name>* <sort>}* <end>

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 however doesn't have any right to modify it.

2.6.2 Start

Abstract grammar

Process-start-node :: *Transition*

Concrete textual grammar

<start> ::=

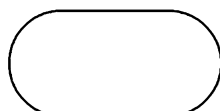
start [<virtuality>] <end> <transition>

Concrete graphical grammar

<start area> ::=

<start symbol>
contains { [<virtuality>] }
is followed by <transition area>

<start symbol> ::=



Semantics

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

Model

A <start> or <start area> which contains <virtuality> is called a virtual start. Virtual start is further described in 6.3.3.

2.6.3 State

Abstract grammar

State-node :: *State-name*
Save-signalset
Input-node-set
Spontaneous-transition-set

State Name = *Name*

State-nodes within a *Process-graph*, *Service-graph* or *Procedure-graph* respectively must have different *State-names*.

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-identifiers* in the *Input-node-set* must be distinct.

Concrete textual grammar

<state> ::=

```
state <state list> <end>
    {   <input part>
    |   <priority input>
    |   <save part>
    |   <spontaneous transition>
    |   <continuous signal> } *
[endstate [<state name>] <end>]
```

<state list> ::=

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

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. For each *State-node*, a *Spontaneous-transition* is represented by a <spontaneous transition>.

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 that <state name>.

```

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

```

$$\langle \text{state symbol} \rangle ::=$$


<input association area> ::=

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

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

<spontaneous transition association area> ::=

<solid association symbol> *is connected to*

<spontaneous transition area>

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

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

A <state area> must contain <state name> if it coincides with a <nextstate area>.

Semantics

A state represents a particular condition in which a process may consume a signal instance resulting in a transition. If the state has neither spontaneous transitions nor continuous signals, and there are no signal instances in the input port, otherwise than those mentioned in a save, then the process waits in the state until a signal instance is received.

At any time in a state, which contains *Spontaneous-transitions*, the process may enter into the *Transition* of one of the *Spontaneous-transitions*.

Model

When the <state list> of a certain <state> contains more than one <state name>, 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

<i>Input-node</i>	::	<i>Signal-identifier</i> [<i>Variable-identifier</i>]*
		<i>Transition</i>

$$\textit{Variable-identifier} = \textit{Identifier}$$

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

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

Concrete textual grammar

```

<input part> ::=
    <basic input part>
  | <remote procedure input transition>

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

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

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

```

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.

Commas may be omitted after the last <variable> in <stimulus>.

Concrete graphical grammar

```

<input area> ::=
    <basic input area>
  | <remote procedure input area>

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

<input symbol> ::=
    <plain input symbol>
  | <internal input symbol>

<plain input symbol> ::=

```



An <input area> whose <input list> contains one <stimulus> corresponds to one *Input-node*. Each of the <signal identifier>s or <timer identifier>s 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 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.

The values will be assigned to the variables from left to right. If there is no variable associated with the input for a sort specified in the signal, the value of this sort is discarded. If there is no value associated with a sort specified in the signal, the corresponding variable becomes “undefined”.

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

Signal instances flowing from the environment to a process instance within the system will always carry a PId value different from any in the system.

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.

When one or more of the <variable>s of a certain <stimulus> are <indexed variable>s or <field variable>s, then all the <variable>s are replaced by unique, new, implicitly declared, <variable identifier>s. Directly following the <input part>, a <task> is inserted which in its <task body> contains an <assignment statement> for each of the <variable>s, assigning the value of the corresponding new variable to the <variable>. The values will be assigned in the order from left to right of the list of <variable>s. This <task> becomes the first <action> in the <transition>.

A <basic input part> or <basic input area> which contains <virtuality> is called a virtual input transition. Virtual input transition is further described in 6.3.3.

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*

Concrete textual grammar

```

<save part> ::=
    <basic save part>
  | <remote procedure save>

<basic save part> ::=
    save [<virtuality>] <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 4.7.

Concrete graphical grammar

```

<save area> ::=
    <basic save area>
  | <remote procedure save area>

<basic save area> ::=
    <save symbol> contains { [<virtuality>] <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.

Model

A <basic save part> or <basic save area> which contains <virtuality> is called a virtual save. Virtual save is further described in 6.3.3.

2.6.6 Spontaneous transition

A spontaneous transition specifies a state transition without any signal reception.

Abstract grammar

Spontaneous transition :: *Transition*

Concrete textual grammar

<spontaneous transition> ::=
 input [<virtuality>] <spontaneous designator> <end>
 [<enabling condition>] <transition>

<spontaneous designator> ::=
 none

Concrete graphical grammar

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

Semantics

A spontaneous transition allows the activation of a transition without any stimuli being presented to the process. The activation of a spontaneous transition is independent of the presence of signal instances in the input port of the process. There is no priority between transitions activated by signal reception and spontaneous transitions.

The **sender** expression of the process is **self** after activation of a spontaneous transition.

Model

A <spontaneous transition> or <spontaneous transition area> which contains <virtuality> is called a virtual spontaneous transition. Virtual spontaneous transition is further described in 6.3.3.

2.6.7 Label

Concrete textual grammar

<label> ::=

<connector name> :

<free action> ::=

connection

<transition>

[**endconnection** [<connector name>] <end>]

<body> is used as a convenient non-terminal for rules which apply to processes, services and procedures. <body> is not part of the concrete textual grammar.

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

A label represents the entry point of a transfer of control from the corresponding joins with the same <connector name>s in the same <body>.

Transfer of control is only allowed to labels within the same <body>. The rule for <body>s of a supertype and its specializations is stated in 6.3.1.

If the <transition string> of the <transition> in <free action> is non-empty, the first <action statement> must have a <label> otherwise the <terminator statement> must have a <label>.

If present, the <connector name> ending the <free action> must be the same as the <connector name> in this <label>.

Concrete graphical grammar

<in-connector area> ::=

<in-connector symbol> **contains** <connector name> **is followed by**

<transition area>

<in-connector symbol> ::=



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>.

Semantics

The contained <transition> or <transition area> is in the abstract grammar represented by applying <join> to the <connector name> (see 2.6.8.2.2)

2.6.8 Transition

2.6.8.1 Transition body

Abstract grammar

<i>Transition</i>	::	<i>Graph-node*</i> (<i>Terminator</i> <i>Decision-node</i>)
<i>Graph-node</i>	::	<i>Task-node</i> <i>Output-node</i> <i>Create-request-node</i> <i>Call-node</i> <i>Set-node</i> <i>Reset-node</i>
<i>Terminator</i>	::	<i>Nextstate-node</i> <i>Stop-node</i> <i>Return-node</i>

Concrete textual grammar

<transition> ::=

	{<transition string> [<terminator statement>] }
	<terminator statement>

<transition string> ::=

{<action statement>}+

<action statement> ::=

[<label>] <action> <end>

<action> ::=

	<task>
	<output>
	<create request>
	<decision>
	<transition option>
	<set>
	<reset>
	<export>
	<procedure call>
	<remote procedure call>

<terminator statement> ::=

[<label>] <terminator> <end>

<terminator> ::=

	<nextstate>
	<join>
	<stop>
	<return>

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 when a <transition> is contained in a <decision> or <transition option>.

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 area>
| <transition option area> }

<transition string area> ::=

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

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, with a return or with the transfer of control to another transition.

2.6.8.2 Transition terminator

2.6.8.2.1 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*, *Service-graph* or *Procedure-graph*.

Concrete textual grammar

<nextstate> ::=

nextstate <nextstate body>

<nextstate body> ::=

{<state name> | <dash nextstate>}

Concrete graphical grammar

<nextstate area> ::=

<state symbol> *contains* <nextstate body>

Semantics

A nextstate represents a terminator of a transition. It specifies the state of the process, service or procedure when terminating the transition.

2.6.8.2.2 Join

A join alters the flow in a <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 exactly one <connector name> corresponding to a <join> within the same <body>. The rule for <process type body> is stated in 6.3.1.

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>, there must be exactly one <in-connector area> in that <process graph area>. The rule for <process type graph area> is stated in 6.3.1.

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>.

Semantics

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

2.6.8.2.3 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 or service interpreting it.

In case of a process, 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.

In case of a service, this means that the signals for the service will be discarded, until the process instance ceases to exist.

2.6.8.2.4 Return

Abstract grammar

Return-node ::= ()

A *Return-node* must be contained in a *Procedure-graph*.

Concrete textual grammar

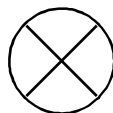
<return> ::= **return** [<expression>]

Concrete graphical grammar

<return area> ::=

<return symbol>
[*is associated with* <expression>]

<return symbol> ::=



<expression> in <return> or <return area> is allowed if and only if the enclosing scope is an operator or a procedure that has <procedure result>.

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) The interpretation of the *Procedure-graph* is completed and the procedure instance ceases to exist.
- c) Hereafter, the calling process, service (or procedure) interpretation continues at the node following the call.

Model

The model for return with an expression is defined in 2.4.6.

2.7 Action

2.7.1 Task

Abstract grammar

<i>Task-node</i>	::	<i>Assignment-statement</i>
		<i>Informal-text</i>

Concrete textual grammar

 $\langle \text{task} \rangle ::=$

task <task body>

$$\langle \text{task body} \rangle ::=$$

```
{<assignment statement>{,<assignment statement>}*}
{<informal text> {,<informal text>}*}
```

Concrete graphical grammar

$$\langle \text{task area} \rangle ::=$$

<task symbol> *contains* <task body>

$$\langle \text{task symbol} \rangle ::=$$


Semantics

The interpretation of a *Task-node* is the interpretation of the *Assignment-statement* or the interpretation of the *Informal-text*.

Model

A <task body> 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 in which they were specified in the <task body> is retained.

This shorthand is expanded before shorthands in the contained expressions are expanded.

2.7.2 Create

Abstract grammar

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

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

Each *Expression* corresponding by position to a *Process-formal-parameter* must have the same sort as the *Process-formal-parameter* in the *Process-definition* denoted by *Process-identifier*.

Concrete textual grammar

<create request> ::=

create <create body>

<create body> ::=

{ <process identifier> | **this** } [<actual parameters>]

<actual parameters> ::=

([<expression>] {,<expression>}*)

Commas after the last <expression> in <actual parameters> may be omitted.

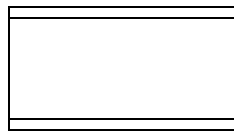
this may only be specified in a <process type definition> and in scopes enclosed by a <process type definition>.

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

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 the same unique, new PId value (see Annex D, D.10.1).

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 5.4.3) to the corresponding formal parameters. 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.

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.

If an <expression> in <actual parameters> is omitted, the corresponding formal parameter has no value associated, i.e. it is “undefined”.

Model

Stating **this** is derived syntax denoting the implicit <process identifier> for the set of instances of which the process executing the create is a member.

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** *Procedure-formal-parameter* must have the same sort as the *Procedure-formal-parameter*.

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

Concrete textual grammar

<procedure call> ::= **call** <procedure call body>

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

An example of <procedure call> is given in Figure 2.10.13.

An <expression> in <actual parameters> corresponding to a formal **in/out** parameter cannot be omitted and must be a <variable identifier>.

After the *Model* for **this** has been applied, the <procedure identifier> must denote a procedure which contains a start transition.

If **this** is used, <procedure identifier> must denote an enclosing procedure.

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 2.10.14.

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 interpretation of the transition containing the procedure call continues when the interpretation of the called procedure is finished.

The actual parameter expressions are interpreted in the order given.

Special semantics are needed as far as data and parameters interpretation are concerned (the explanation is contained in 2.4.6).

If an <expression> in <actual parameters> is omitted, the corresponding formal parameter has no value associated, i.e. it is “undefined”.

Model

If the <procedure identifier> is not defined within the enclosing service or process, the procedure call is transformed into a call of a local, implicitly created, subtype of the procedure.

this implies that when the procedure is specialized, the <procedure identifier> is replaced by the identifier of the specialized procedure.

2.7.4 Output

Abstract grammar

<i>Output-node</i>	::	<i>Signal-identifier</i> [<i>Expression</i>]* [<i>Signal-destination</i>] <i>Direct-via</i>
<i>Signal-destination</i>	=	<i>Expression</i> <i>Process-identifier</i>
<i>Direct-via</i>	=	(<i>Signal-route-identifier</i> <i>Channel-identifier</i>)-set

The length of [*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 clause 3) there must exist at least one communication path (either implicit to own set of process instances, or explicit via signal routes and possibly channels) to the environment, or to a process instance set or service having *Signal-identifier* in its valid input signal set and originating from the process instance set or service where the *Output-node* is used.

For each *Signal-route-identifier* in *Direct-via*, the *Origin* in (one of) the *Signal-route-path(s)* of the signal route must denote either

- a set of process instances that contains the process interpreting the *Output-node* or containing a service that interprets the *Output-node*, or
- a service within the originating process that interprets the *Output-node*.

Further, the *Signal-route-path* must include *Signal-identifier* in its set of *Signal-identifiers*. If the *Origin* denotes a set of process instances and the signal is sent from a service, then there must exist a signal route within the process capable of conveying the signal from the sending service to the signal route mentioned in *Direct-via*.

For each *Channel-identifier* in *Direct-via* there must exist one or two signal routes and zero or more channels such that the channel via this path is reachable with the *Signal-identifier* from the process or service, and the *Channel-path* in the direction from the process must include *Signal-identifier* in its set of *Signal-identifiers*.

Concrete textual grammar

<output> ::=

output <output body>

<output body> ::=

<signal identifier>

[<actual parameters>]{, <signal identifier> [<actual parameters>]}*

[**to** <destination>] [**via** [**all**] <via path>]

<destination> ::=

<PId expression> | <process identifier> | **this**

<via path> ::=

<via path element> {, <via path element>}*

<via path element> ::=

<signal route identifier>

| <channel identifier>

| <gate identifier>

The <PId expression> or the <process identifier> in <destination> represents the *Signal-destination*. There is a syntactic ambiguity between <PId expression> and <process identifier> in <destination>. If <destination> can be interpreted as a <PId expression> without violating any static conditions, it is interpreted as <PId expression> otherwise as <process identifier>. <process identifier> must denote a process, which is reachable from the originating process.

The **via** construct represents the *Direct-via*. The <via path> is considered a list of <via path element>s.

via all may only be specified when there is no *Signal-destination*. **via all** represents multicast as explained in *Model*.

this may only be specified in a <process type definition> and in scopes enclosed by a <process type definition>.

Concrete graphical grammar

<output area> ::=

<output symbol> **contains** <output body>

<output symbol> ::=

<plain output symbol>

| <internal output symbol>

<plain output symbol> ::=



Semantics

Stating a *Process-identifier* in *Signal-destination* indicates *Signal-destination* as any existing instance of the set of process instances indicated by *Process-identifier*. If there exist no instances, the signal is discarded.

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

The values conveyed by the signal instance are the values of the actual parameters of the output. If an <expression> in <actual parameters> is omitted, no value is conveyed with the corresponding place of the signal instance, i.e. the corresponding place is “undefined”.

The PId value of the originating process is also conveyed by the signal instance.

If a syntype is specified in the signal definition and an expression is specified in the output, then the range check defined in 5.3.1.9.1 is applied to the expression.

The signal instance is then delivered to a communication path able to convey it. The set of communication paths able to convey the signal instance can be restricted by the **via** clause to the set of paths mentioned in the *Direct-via*.

If *Signal-destination* is *Expression*, the signal instance is delivered to the process instance denoted by *Expression*. If this instance does not exist or is not reachable from the originating process, the signal instance is discarded.

If *Signal-destination* is *Process-identifier*, the signal instance is delivered to an arbitrary instance of the process instance set denoted by *Process-identifier*. If no such instance exists, the signal instance is discarded.

For example, if *Signal-destination* is specified as Null in an *Output-node*, the signal instance will be discarded, when the *Output-node* is interpreted.

If no *Signal-destination* is specified, the receiver is selected in two steps. First, the signal is sent to a process instance set, which can be reached by the communication paths able to convey the signal instance. This process instance set is arbitrarily chosen. Second, when the signal instance arrives at the end of the communication path, it is delivered to an arbitrary instance of the process instance set. The instance is arbitrarily selected. If no instance can be selected, the signal instance is discarded.

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 delaying channels, or only conveyed by channels with no delay, or if the originating process or service and the destination process or service are defined within the same block.

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.

Stating **via all** is derived syntax for multicasting the signal via the communication paths mentioned in <via path>, so that signals are sent in the same order as <via path element>s appear in <via path>, one via each <via path element>. Multicast denotes a sequence of outputs of the same signal. The values conveyed with each of the resulting signal instances are only evaluated once before interpreting the first output, implicit variables are then used to store the values for use in each output. If same <via path element> appears several times in a <via path>, one signal is sent for each appearance.

Stating **this** is derived syntax for denoting as <process identifier>, the implicit <process identifier> for the set of instances of which the process executing the output is a member.

2.7.5 Decision

Abstract grammar

<i>Decision-node</i>	::	<i>Decision-question</i> <i>Decision-answer-set</i> <i>[Else-answer]</i>
<i>Decision-question</i>	=	<i>Expression</i> <i>Informal-text</i>
<i>Decision-answer</i>	::	<i>(Range-condition Informal-text)</i> <i>Transition</i>
<i>Else-answer</i>	::	<i>Transition</i>

The *Range-conditions* of the *Decision-answers* must be mutually exclusive, and the *Ground-Expressions* of the *Range-conditions* must be of the same sort.

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> ::=	< <u>question</u> expression> <informal text> any

A <decision> or <transition option> is a terminating decision and terminating option respectively, if each <answer part> and <else part> in its <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.

The <answer> of <answer part> must be omitted if and only if the <question> consists of the keyword **any**. In this case, no <else part> may be present.

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>.

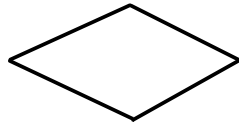
The context of the decision (i.e. the sort) is determined without regard to <answer>s which are <character string>.

If a <character string> of <question> or any <answer> contain control characters, the string is interpreted as <informal text>.

<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>

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*.

The <answer> of <graphical answer> must be omitted if and only if the <question> consists of the keyword **any**. In this case, no <graphical else part> may be present.

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.

Model

If a <decision> is not terminating 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>.

Using only **any** in a <decision> is a shorthand for using <anyvalue expression> in the decision. Assuming that the <decision body> is followed by N <answer part>s, **any** in <decision> is then a shorthand for writing **any**(data_type_N), where data_type_N is an anonymous syntype defined as

```

syntype data_type_N =
  package Predefined Integer constants 1:N
endsyntype;

```

The omitted <answer>s are shorthands for writing the literals 1 through N as the <constant>s of the <range condition>s in the N <answer>s.

2.8 Timer

Abstract grammar

<i>Timer-definition</i>	::	<i>Timer-name</i> <i>Sort-reference-identifier</i> *
<i>Timer-name</i>	=	<i>Name</i>
<i>Set-node</i>	::	<i>Time-expression</i> <i>Timer-identifier</i> <i>Expression</i> *
<i>Reset-node</i>	::	<i>Timer-identifier</i> <i>Expression</i> *
<i>Timer-identifier</i>	=	<i>Identifier</i>
<i>Time-expression</i>	=	<i>Expression</i>

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*.

Concrete textual grammar

```

<timer definition> ::=
    timer
    <timer definition item> { , <timer definition item> }* <end>

<timer definition item> ::=
    <timer name> [ <sort list> ] [ := <Duration ground expression> ]

<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> ) ] )

```

A <set statement> may omit <Time expression>, if <timer identifier> denotes a timer which has a <Duration ground expression> in its definition.

A <reset statement> represents a *Reset-node*; a <set statement> represents a *Set-node*.

Concrete graphical grammar

```

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

```

<reset area> ::=

<task symbol> *contains* <reset>

Semantics

A timer instance is an object, 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 operator “=” applied to all corresponding expressions in the lists yields True (i.e. 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 less than or equal to **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 5.3.19.1 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.

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

Model

A <set statement> with no <Time expression> is derived syntax for a <set statement> where <Time expression> is “**now** + <Duration ground expression>”, where <Duration ground expression> is derived from the timer definition.

A <reset> or a <set> may contain several <reset statement>s or <set statement>s respectively. This is derived syntax for specifying a sequence of <reset>s or <set>s, one for each <reset statement> or <set statement> such that the original order in which they were specified in <reset> or <set> is retained. This shorthand is expanded before shorthands in the contained expressions are expanded.

2.9 Internal input and output

The symbols defined in this section are introduced for compatibility with existing diagrams. They are not recommended for new SDL descriptions and they have the same semantics as the <plain input symbol> and <plain output symbol> respectively.

Concrete graphical grammar

<internal input symbol> ::=



<internal output symbol> ::=



2.10 Examples

```
task t3 := 0/*example*/;
```

FIGURE 2.10.1/Z.100

Example of note (SDL/PR)

```
-----  
task 'task1' comment 'example';  
task 'task2';  
-----
```

FIGURE 2.10.2/Z.100

Example of comment (SDL/PR)

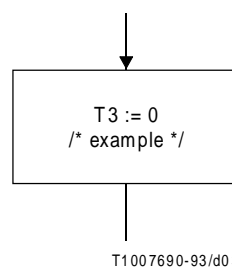


FIGURE 2.10.3/Z.100

Same example as Figure 2.10.1 in SDL/GR

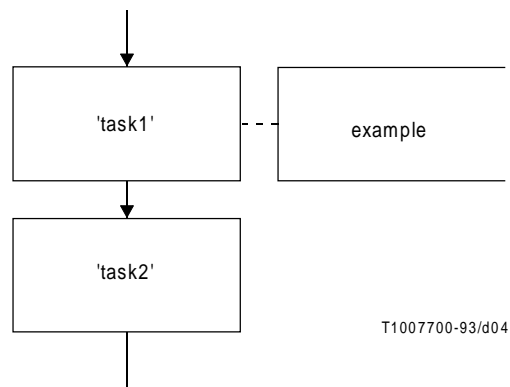


FIGURE 2.10.4/Z.100
Same example as Figure 2.10.2 in SDL/GR

```

system Daemongame;

  signal Newgame, Probe, Result, Endgame, Gameid, Win, Lose,
    Score (Integer);

  channel Gameserver.in

    from env to Game

    with Newgame, Probe, Result, Endgame;

  endchannel Gameserver.in;

  channel Gameserver.out

    from Game to env

    with Gameid, Win, Lose, Score;

  endchannel Gameserver.out;

  block Game referenced;

endsystem Daemongame;

```

FIGURE 2.10.5/Z.100
Example of a system specification (SDL/PR)

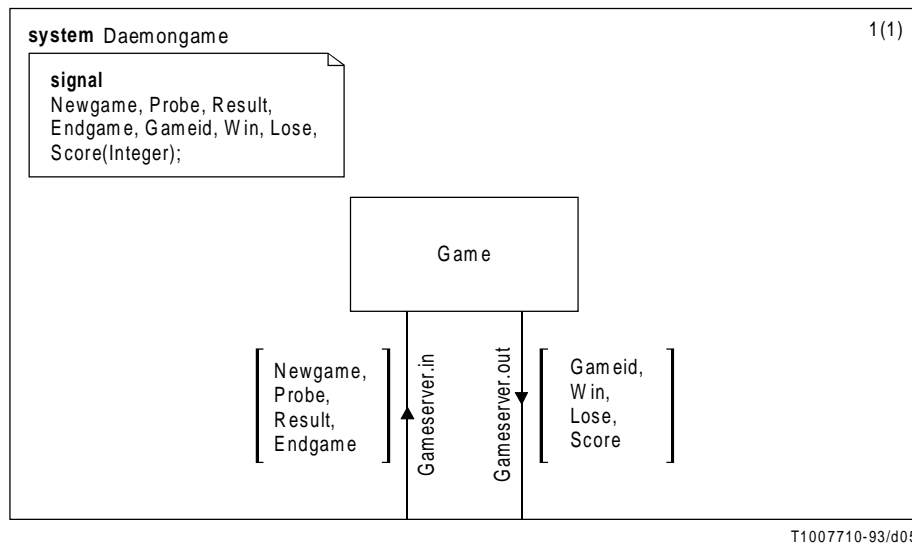


FIGURE 2.10.6/Z.100
Example of a system specification (SDL/GR)

```

block Game;
    signal Gameover(PId);
    connect Gameserver.in and R1,R2;
    connect Gameserver.out and R3;
    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 Monitor with Gameover;

    process Monitor (1,1) referenced;

    process Game (0,) referenced;
endblock Game;

```

FIGURE 2.10.7/Z.100
Example of block specification (SDL/PR)

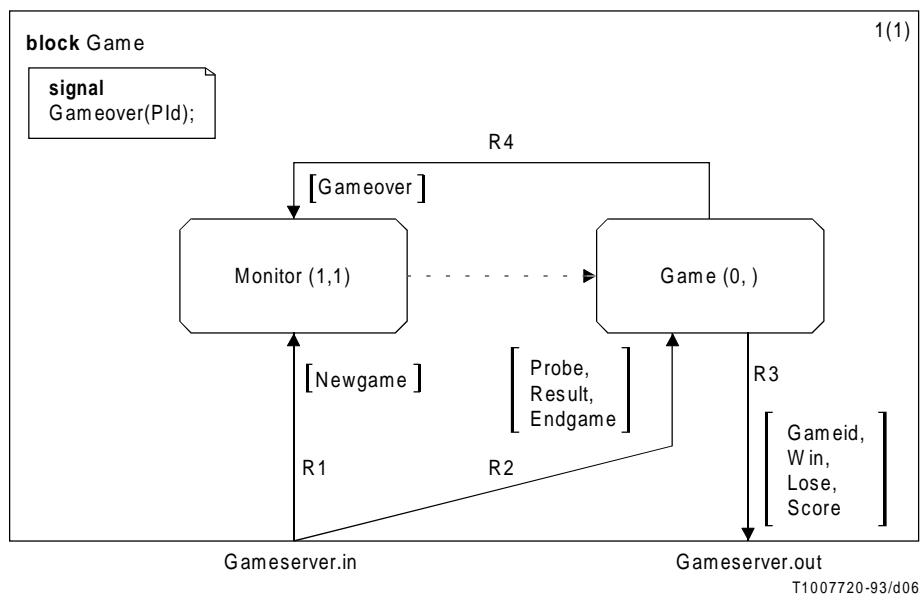


FIGURE 2.10.8/Z.100
Example of a block specification (SDL/GR)

```

process Game (0, ); fpar player Pid;

dcl count Integer := 0 ; /*counter to keep track of score */

start;
    output Gameid to player;
    nextstate even;

state even;
    input none;
    nextstate odd;
input Probe;
    output Lose to player;
    task count:= count-1;
    nextstate -;

state odd;
    input Probe;
    output Win to player;
    task count:= count+1;
    nextstate -;
input none;
    nextstate even;

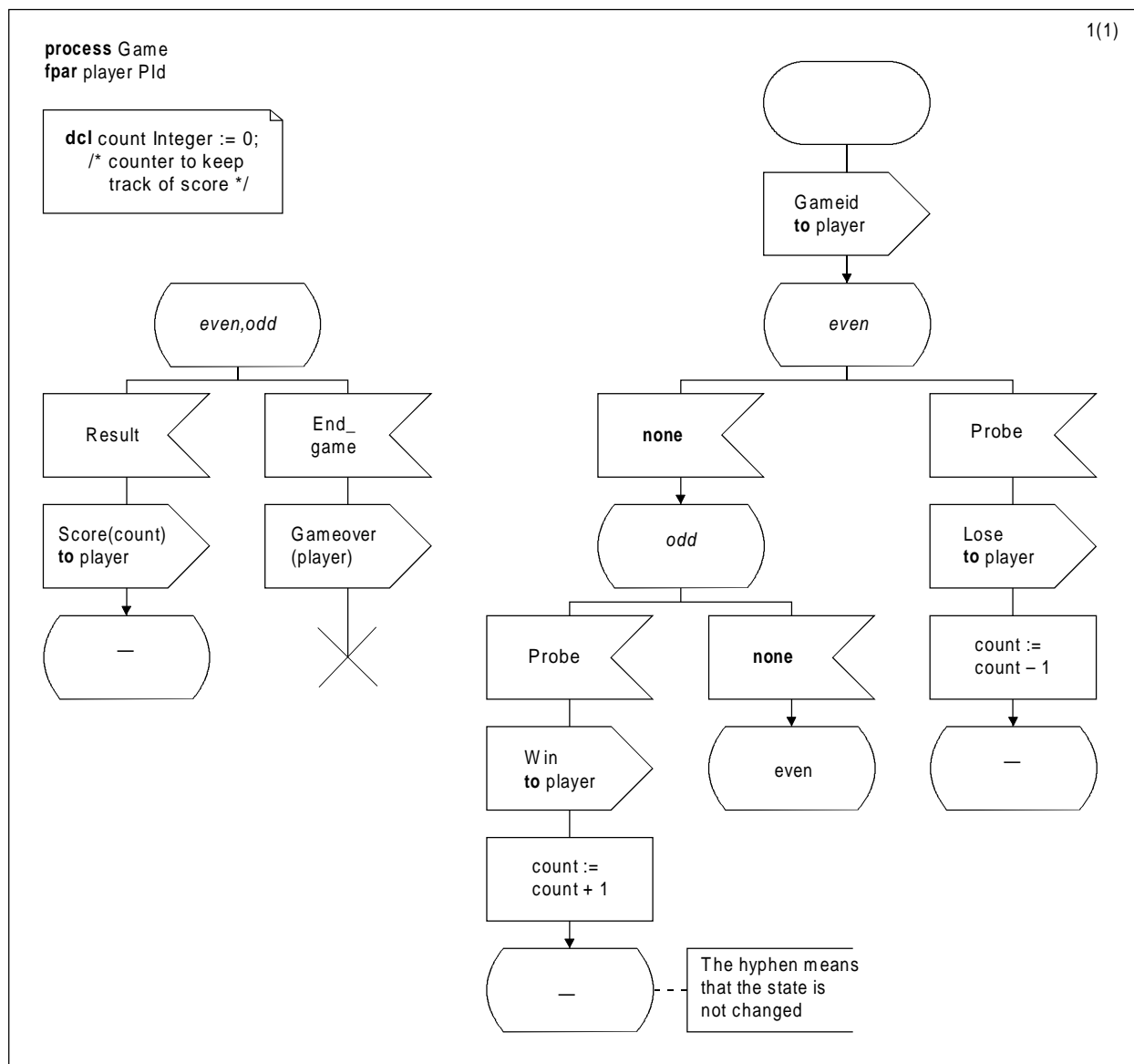
state even, odd;
    input Result;
    output Score(count) to player;
    nextstate -;
input Endgame;
    output Gameover(player);
    stop;

endprocess Game;

```

FIGURE 2.10.9/Z.100

Example of process specification (SDL/PR)



T1007730-93/d07

FIGURE 2.10.10/Z.100

Example of process specification (SDL/GR)

```

procedure Check;
fpar in/out x, y Integer;
/* The following signal definitions are assumed:
signal sig1(Boolean), sig2, sig3(Integer, PId);    */
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 problems;
    enddecision;
    input sig2;
    join 1;
    .....
endprocedure Check;
    
```

FIGURE 2.10.11/Z.100

Example of a fragment of a procedure specification (SDL/PR)

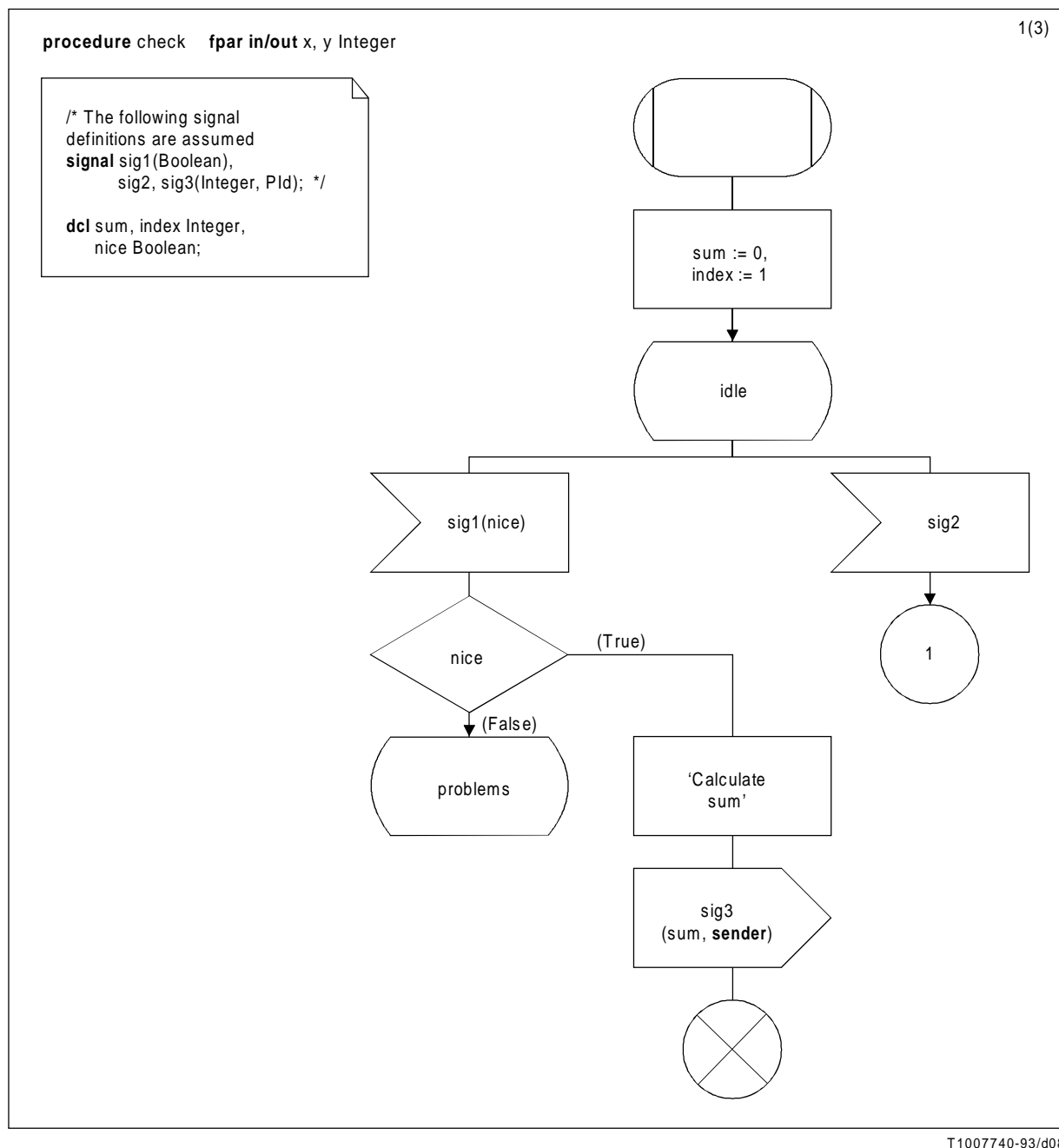


FIGURE 2.10.12/Z.100

Example of a fragment of a procedure specification (SDL/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 2.10.13/Z.100

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

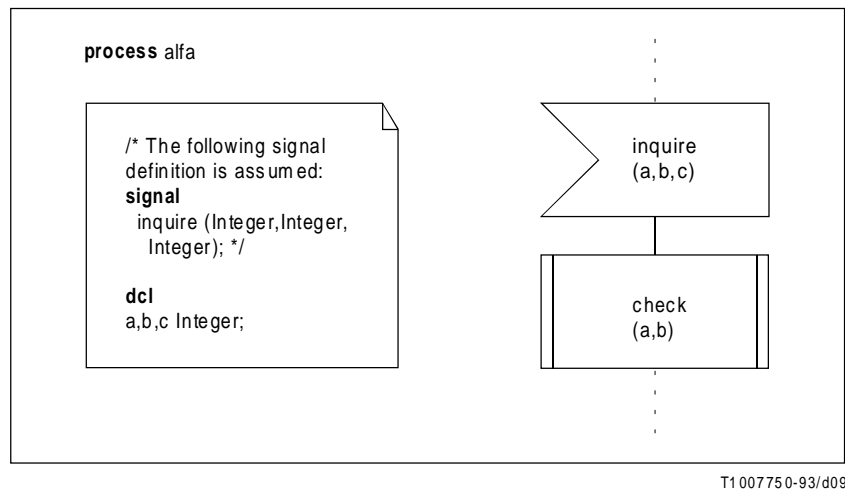


FIGURE 2.10.14/Z.100
 Example of a procedure call in a fragment
 of a process definition (SDL/GR)

An example of a <process definition> containing a <service definition>s is given below in Figures 2.10.15 to 2.10.20 with the corresponding <service definition>s. This process has the same behaviour as the one given in Figures 2.10.9 and 2.10.10.

```
process Game;
  fpar player Pid;
  signal Proberes (Integer);

  signalroute IR1 from Game_handler to env with Score,Gameid;
  signalroute IR2 from Game_handler to env with Gameover;
  signalroute IR3 from env to Game_handler with Result,Endgame;
  signalroute IR4 from env to Probe_handler with Probe;
  signalroute IR6 from Probe_handler to env with Lose,Win;
  signalroute IR7 from Probe_handler to Game_handler with Proberes;

  connect R2 and IR3,IR4;
  connect R3 and IR1,IR6;
  connect R4 and IR2;

  service Game_handler referenced;
  service Probe_handler referenced;

endprocess Game;
```

FIGURE 2.10.15/Z.100
 Example of process (same as Figure 2.10.9) decomposed as services (SDL/PR)

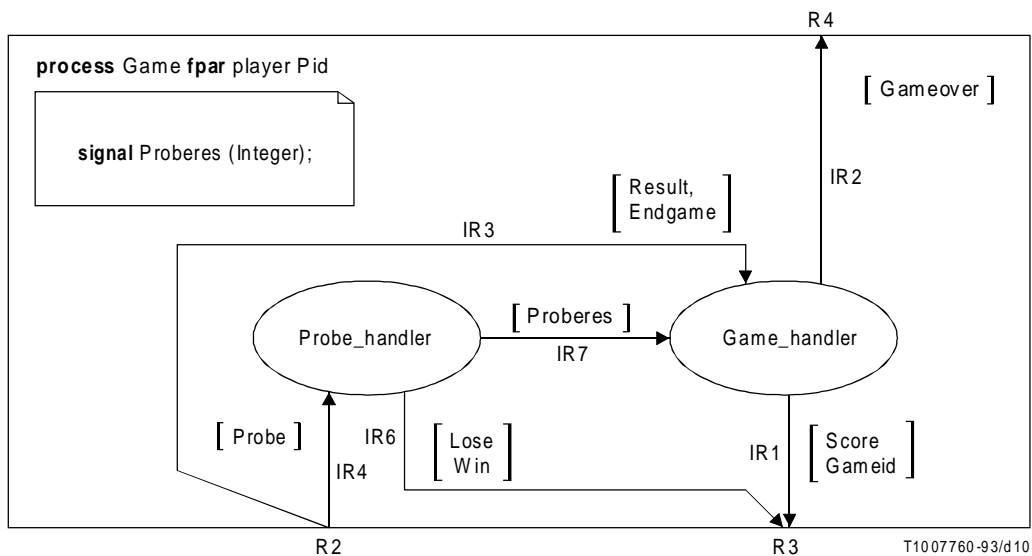


FIGURE 2.10.16/Z.100
Example of process (same as FIGURE 2.10.10)
decomposed as services (SDL/GR)

```

service Game_handler;

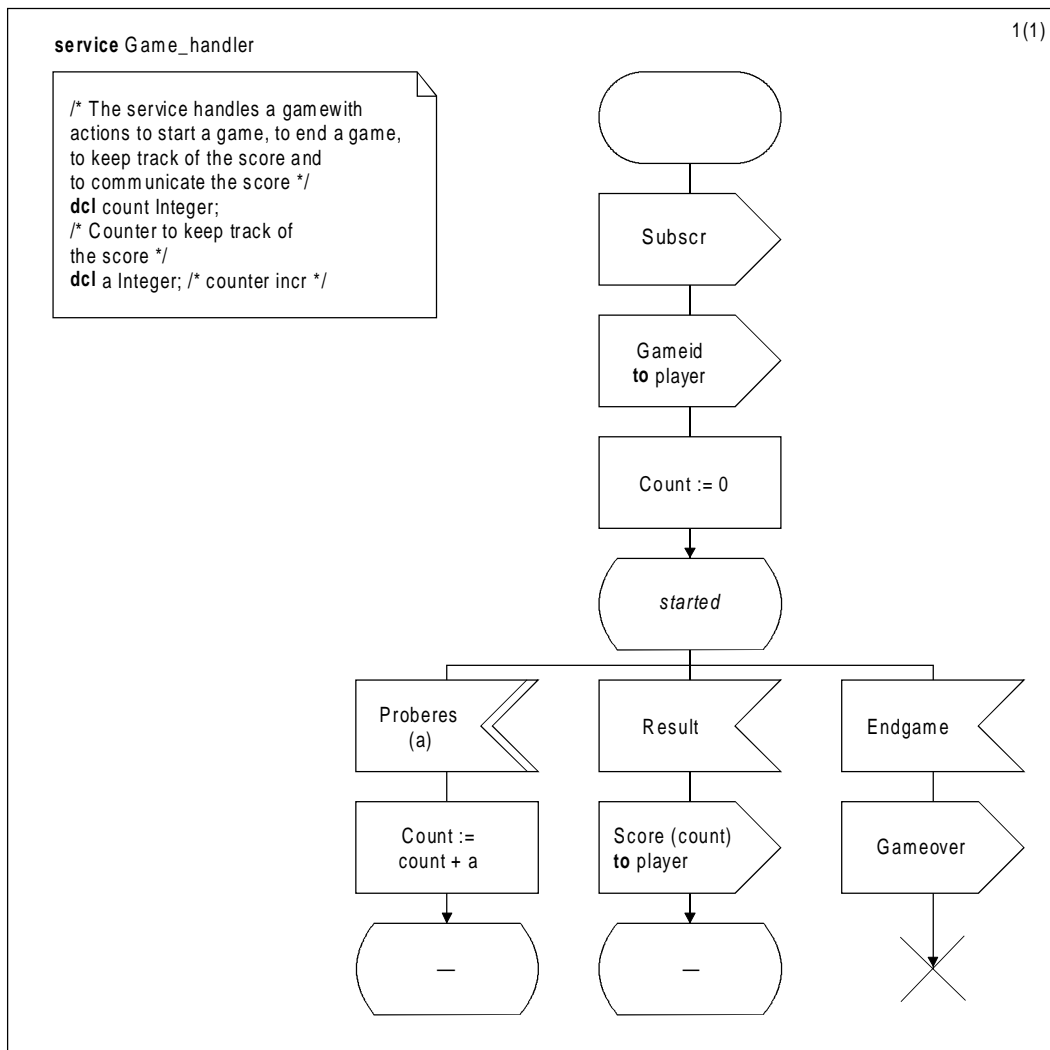
/* The service handles a game with actions to start a game, to end
a game, to keep track of the score and to communicate the score */

dcl  count Integer,    /*Counter to keep track of the score*/
      a Integer;       /* counter increment */

start;
  output Subscr;
  output Gameid to player;
  task count:=0;
  nextstate started;
state started;
  priority input Proberes(a);
  task count:=count+a;
  nextstate -;
  input Result;
  output Score(count) to player;
  nextstate -;
  input Endgame;
  output Gameover;
  stop;
endstate started;
endservice Game_handler;

```

FIGURE 2.10.17/Z.100
Example of service (SDL/PR)



T1007770-93/d11

FIGURE 2.10.18/Z.100

Example of service (same as FIGURE 2.10.17) (SDL/PR)

```

service Probe_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;
    output Proberes(-1) to self;
    nextstate -;
  input none;
    nextstate Odd;
endstate even;
state odd;
  input none;
    nextstate even;
  input Probe;
    output Win to player;
    output Proberes(1) to self;
    nextstate -;
endstate Odd;
endservice Bump_handler;
  
```

FIGURE 2.10.19/Z.100

Example of service (SDL/PR)

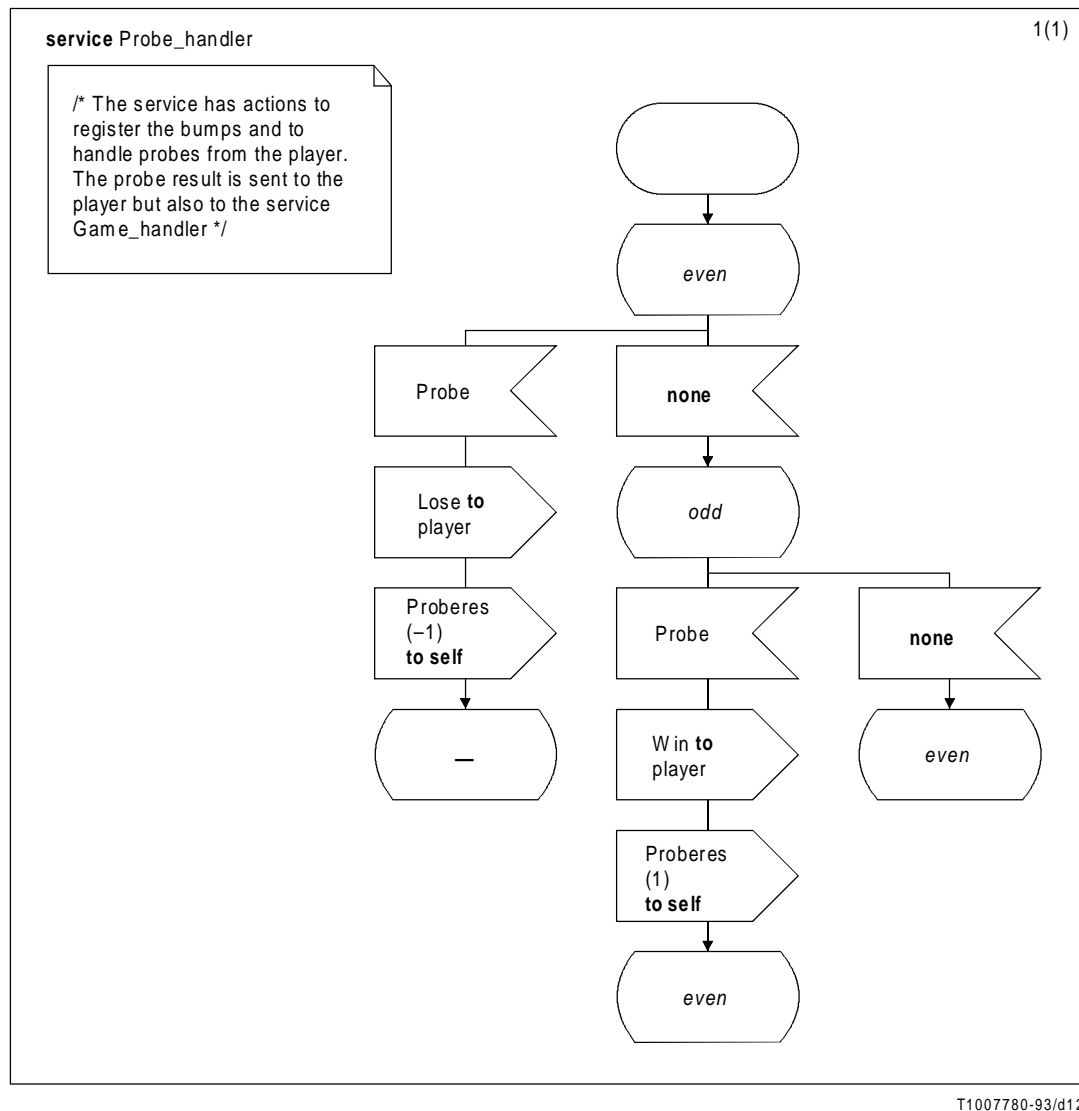


FIGURE 2.10.20/Z.100
Example of service (same as FIGURE 2.10.19) (SDL/GR)

3 Structural Decomposition 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 clause 2. The concepts defined here are strictly additions to those defined in clause 2.

The intention of the concepts introduced in this section is to provide the user with the means to specify large and/or complex systems. The concepts defined in clause 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 dealt with 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. In addition to concepts introduced in this section, the package concept introduced in clause 2 and the structural typing concepts introduced in clause 6 are particularly useful for describing large systems.

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 may be partitioned into a set of subblocks, channels and subchannels. Similarly, a channel may be partitioned into a set of blocks, channels and subchannels. In the concrete syntaxes, each block (or block type) definition and channel definition can have two versions: an unpartitioned version and a partitioned version. However, channel substructures are transformed when mapping onto the abstract syntax, i.e. a channel with a substructure is never interpreted, instead the channel substructure is interpreted. 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-definition* 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 must contain *Process-definitions*.

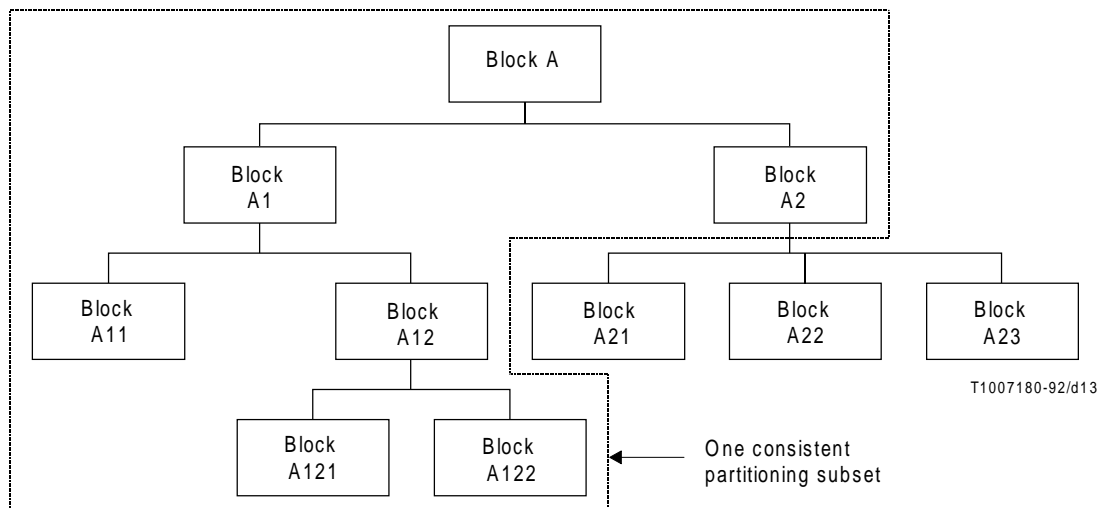


FIGURE 3.2.1/Z.100
Consistent partitioning subset illustrated in an auxiliary diagram

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
Syntype-definition-set

Block-substructure-name = *Name*

Sub-block-definition = *Block-definition*

Channel-connection :: *Channel-identifier-set*
Sub-channel-identifier-set

Sub-channel-identifier = *Channel-identifier*

Channel-identifier = *Identifier*

The *Block-substructure-definition* must contain at least one *Sub-block-definition*. Unless stated otherwise, it is understood in the following that an abstract syntax term is contained in the *Block-substructure-definition*.

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

Each *Channel-definition* in the enclosing block definition that is connected to a *Block-substructure-definition* must be a member of exactly one *Channel-connection*. The *Channel-identifiers* in the *Channel-connection* must identify

Channel-definitions in the enclosing block definition. Each *Sub-channel-identifier* must appear in exactly one *Channel-connection*.

For signals directed out of the *Block-substructure-definition*, the union of the *Signal-identifier-sets* associated with the *Sub-channel-identifier-set* contained in a *Channel-connection* must be identical to the union of the *Signal-identifier-sets* associated with the *Channel-identifier-set* 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.

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 Figure 3.2.2 below.

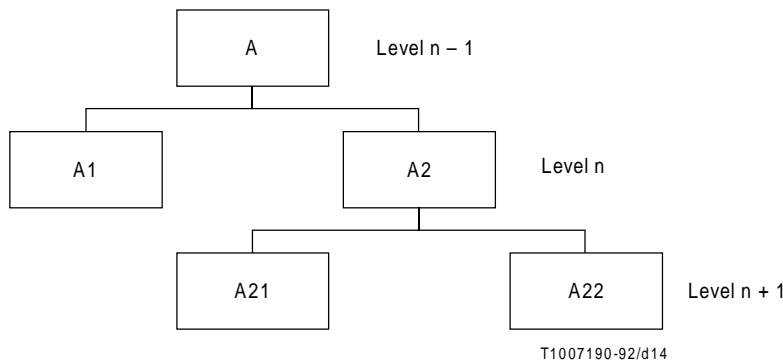


FIGURE 3.2.2/Z.100

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>

{ <entity in system> | <channel connection> }+

endsubstructure

[{<block substructure name> | <block substructure identifier>}] <end>

If the <block substructure name> after the keyword **substructure** is omitted, it is the same as the name of the enclosing <block definition> or <block type definition>.

<textual block substructure reference> ::=

substructure <block substructure name> **referenced** <end>

<channel connection> ::=

connect <channel identifiers>

and <subchannel identifiers> <end>

<subchannel identifiers> ::=

<channel identifiers>

If a <block substructure definition> contains <channel definition>s and <textual typebased block definition>s, then each gate of the <block type definition>s of the <textual typebased block definition>s must be connected to a channel.

For a <block substructure definition> in a <block type definition>, <channel connection>s cannot be given. These are derived for the resulting <textual typebased block definition>s as defined in 6.1.4.

Concrete graphical grammar

```
<block substructure diagram> ::=
    <frame symbol>
    contains {<block substructure heading>
        { {<block substructure text area>}*
            {<macro diagram>}*
            <block interaction area>
            {<type in system area> }*}set }
    is associated with {<channel identifiers>}*
```

The <channel identifiers> identify channels connected to subchannels in the <block substructure diagram>. They are placed outside the <frame symbol> close to the endpoint of the subchannels at the <frame symbol>.

A <channel symbol> within the <frame symbol> and connected to the <frame symbol> 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> or <block type diagram> must not contain other definitions than formal context parameters, gates and the substructure definition.

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> is the same as the <name> of the enclosing <block diagram> or <block type diagram>.

Example

Figure 3.3.1 shows an example of an <open block substructure diagram>.

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

```
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;
```

The <block substructure diagram> for the same example is given in Figure 3.2.3 below.

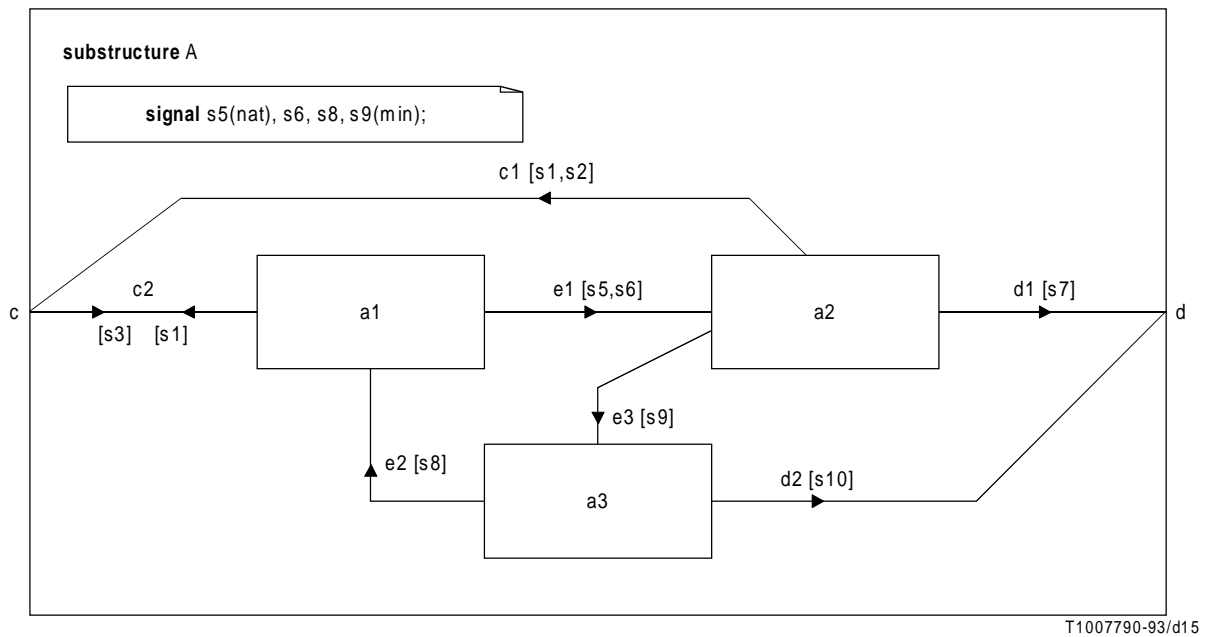


FIGURE 3.2.3/Z.100
Block substructure diagram for block A

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>  
{  
| <entity in system>  
| <channel endpoint connection> }+  
endsubstructure [{ <channel substructure name>  
| <channel substructure identifier>}] <end>
```

If the <channel substructure name> after the keyword **substructure** is omitted, it is the same as the <channel name> of the enclosing <channel definition>.

<textual channel substructure reference> ::=

```
substructure <channel substructure name> referenced <end>
```

<channel endpoint connection> ::=

```
connect {<block identifier> | env}  
and <subchannel identifiers> <end>
```

The two endpoints of the partitioned <channel definition> must be different, and for each endpoint there must be exactly one <channel endpoint connection>. The <block identifier> or **env** in a <channel endpoint connection> must identify one of the endpoints of the partitioned <channel definition>.

Additional static conditions of a <channel substructure definition> are defined in terms of the <block definition> which results from the transformation described in *Model*.

Concrete graphical grammar

<channel substructure diagram> ::=

```
<frame symbol>  
contains {<channel substructure heading>  
| {<channel substructure text area>}*  
| {<macro diagram>}*  
| {<type in system area>}*  
| <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. The two implicit channels are delaying if and only if the substructured channel is delaying.
- 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.
- c) <output>s within the channel substructure which mention the channel in a <via path> have its via-clause mentioning the <channel identifier> replaced by a via-clause containing one or two of the implicit channels in its <via path>, so that a channel which has the <signal identifier> in its <signal list> for a direction from the <block definition> is in the <via path>. In case the <channel identifier> is replaced by two implicit channels in <via path>, these occur in arbitrary order.

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 in Figure 3.2.4 below.

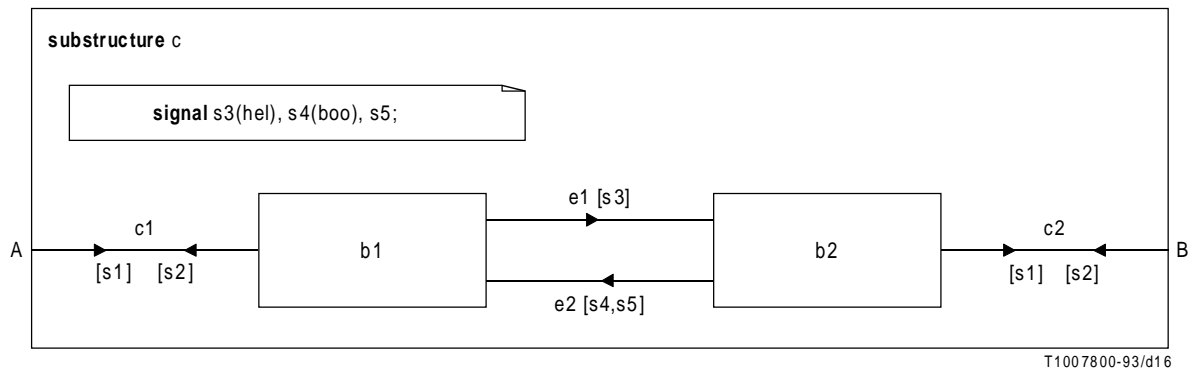


FIGURE 3.2.4/Z.100
Channel substructure diagram for channel C

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 (direct or indirect) subsignal definitions. A subsignal definition of a signal definition is not 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 with each *Channel-identifier* either the *Signal-identifier* is associated with at least one of the *Sub-channel-identifiers*, or each of its subsignal identifiers is associated with 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 set of process instances denoted by 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>
```

```
<reverse> ::=
    [reverse]
```

A <signal definition> in a <subsignal definition> cannot have formal context parameters.

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 connected to subchannels in a substructure. 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 a direction opposite to the refined signal, which is indicated by the keyword **reverse**. Signals can only be refined at <channel connection>s and at <channel endpoint connection>s.

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 (as defined in 3.2.1) restricted by the additional rule:

- d) When selecting the consistent partitioning subset, signals must be used on same refinement level in communicating processes. That is, for each process instance set which via communications paths may convey a signal to another process instance set, the signal must be used on same refinement level in both sets.

Example

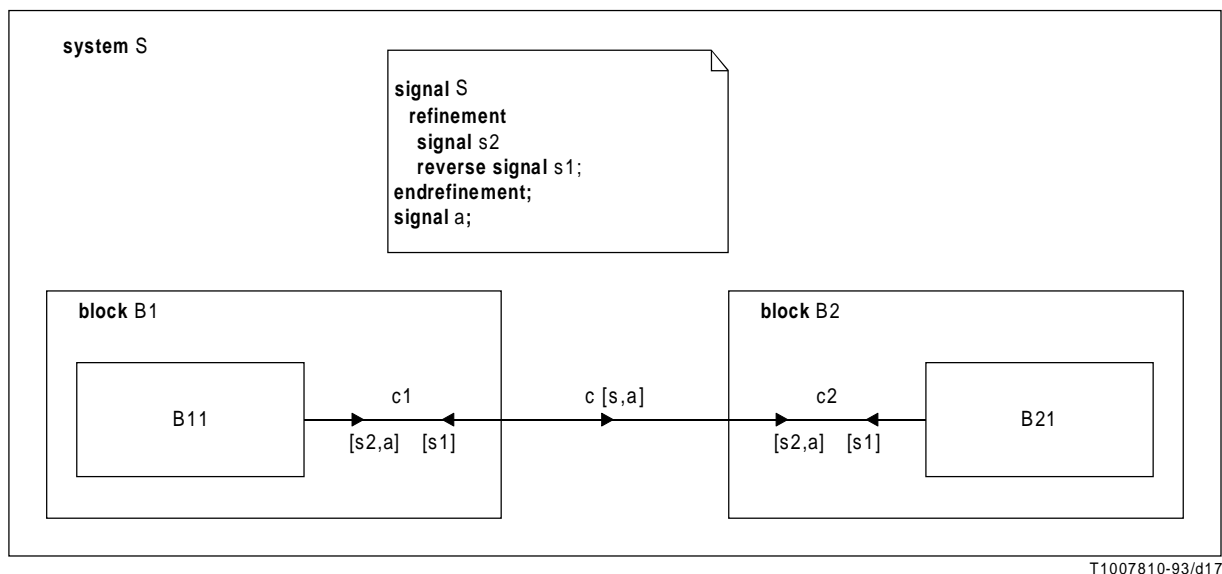


FIGURE 3.3.1/Z.100
System diagram containing signal refinement

In the example given in Figure 3.3.1 above, signal s is refined in block definition B1 and B2, but not signal a. On the highest refinement level, processes in B1 and B2 communicate using signal s and a. On the next lower level, processes in B11 and B21 communicate using s1, s2 and a.

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 of Basic SDL

4.1 Introduction

This chapter defines a number of additional concepts. They are introduced for the convenience of the users in addition to shorthand notations defined in other chapters of the Recommendation.

The properties of a shorthand notation are derived from the way it is modelled 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 defined in clause 7. The transformation order is also followed when defining the concepts in this section.

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 <sdl specification>. Each such place is indicated by a macro call. Before an <sdl specification> 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>}*
    [%<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>]

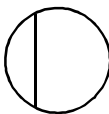
<macro body area> ::=

{ {<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> | <macro outlet symbol> }
 [{ *contains* <macro label>
 | *is associated with* <macro label> }] }

<macro body port2> ::=

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

<block area>
 | <block interaction area>
 | <block substructure area>
 | <block substructure text area>
 | <block text area>
 | <block type reference>
 | <channel definition area>
 | <channel substructure area>

<channel substructure association area>
 <channel substructure text area>
 <comment area>
 <continuous signal area>
 <continuous signal association area>
 <create line area>
 <create request area>
 <decision area>
 <enabling condition area>
 <existing typebased block definition>
 <export area>
 <graphical block reference>
 <graphical typebased block definition>
 <graphical procedure reference>
 <graphical process reference>
 <in-connector area>
 <input area>
 <input association area>
 <macro call area>
 <merge area>
 <nextstate area>
 <operator heading>
 <operator text area>
 <option area>
 <out-connector area>
 <output area>
 <package reference area>
 <package text area>
 <priority input area>
 <priority input association area>
 <procedure area>
 <procedure call area>
 <procedure graph area>
 <procedure start area>
 <procedure text area>
 <process area>
 <process graph area>
 <process interaction area>
 <process text area>
 <process type graph area>
 <process type reference>
 <remote procedure call area>
 <remote procedure input area>
 <remote procedure save area>
 <reset area>
 <return area>
 <save area>
 <save association area>
 <service area>
 <service interaction area>
 <service graph area>
 <service text area>
 <service type reference>
 <set area>
 <signal list area>
 <signal route definition area>
 <spontaneous transition association area>
 <spontaneous transition area>
 <start area>
 <state area>
 <stop symbol>
 <system text area>
 <task area>

	<text extension area>
	<transition area>
	<transition option area>
	<transition string area>
	<type in system area>
	<type in block area>
	<type in process area>

A <dummy outlet symbol> must not have anything associated with 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.

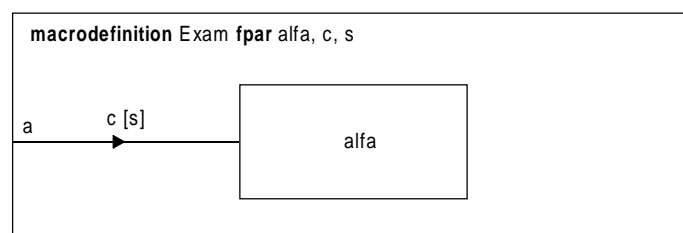
The keyword **macroid** may be used as a pseudo macro formal parameter within each macro definition. No <macro actual parameter>s can be given to it, and it is replaced by a unique <name> for each expansion of a macro definition (within an expansion the same <name> is used for each occurrence of **macroid**).

Example

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

```
macrodefinition Exam
fpar alfa, c, s, a;
  block alfa referenced;
  channel c from a to alfa with s; endchannel c;
endmacro Exam;
```

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



T1007820-93/d18

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 with 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 occurrence 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 must be one to one correspondence between <macro formal parameter> and <macro actual parameter>.

Model

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 body port2> which have no corresponding <macro call port1> or <macro call port2> are also deleted.

Example

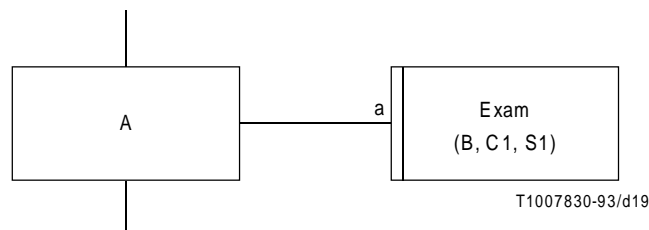
Below is given an example of a <macro call>, within a fragment of a <block definition>.

```
.....
block A referenced;
macro Exam (B, C1, S1, A);
.....
```

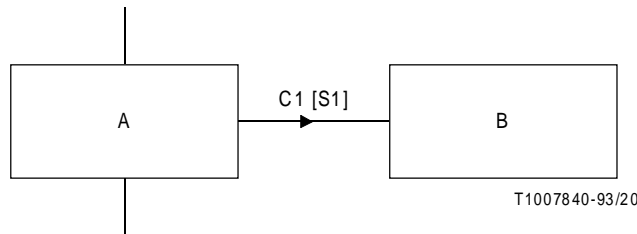
The expansion of this macro call, using the example in 4.2.2, gives the following result.

```
.....
block A referenced;
block B referenced;
channel C1 from A to B with S1; endchannel C1;
.....
```

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 system definition

A system specification may have optional parts and system parameters with unspecified 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 of 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 definition item>
 {, <external synonym definition item> }*

<external synonym definition item> ::=

<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 5.3.1.13. An <external synonym> may be used at any place where a <synonym> is allowed, see 5.3.3.3. The predefined sorts are: Boolean, Character, Charstring, Integer, Natural, Real, PId, Duration and 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>

The *Ground-expression* represented by a <simple expression> must only contain operators and literals defined within the package Predefined, as defined in Annex D.

A simple expression is a *Ground-expression*.

4.3.3 Optional definition

Concrete textual grammar

<select definition> ::=

```

select if (<Boolean simple expression> ) <end>
{
    <system type definition>
    | <textual system type reference>
    | <block type definition>
    | <textual block type reference>
    | <block definition>
    | <textual block reference>
    | <textual typebased block definition>
    | <channel definition>
    | <signal definition>
    | <signal list definition>
    | <remote variable definition>
    | <remote procedure definition>
    | <data definition>
    | <process type definition>
    | <textual process type reference>
    | <process definition>
    | <textual process reference>
    | <textual typebased process definition>
    | <service type definition>
    | <textual service type reference>
    | <service definition>
    | <textual service reference>
    | <textual typebased service definition>
    | <timer definition>
    | <channel connection>
    | <channel endpoint connection>
    | <variable definition>
    | <view definition>
    | <imported variable specification>
    | <procedure definition>
    | <textual procedure reference>
    | <imported procedure specification>
    | <signal route definition>
    | <channel to route connection>
    | <signal route to route connection>
    | <select definition>
    | <macro definition> }+
endselect <end>

```

The only visible names in a <Boolean simple expression> of a <select definition> are names of external synonyms defined outside of any <select definition>s or <option area>s and literals and operators of the sorts defined within the package Predefined as defined in Annex D.

A <select definition> may contain only those definitions that are syntactically allowed at that place.

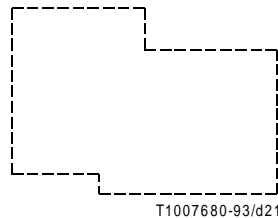
<option area> ::=

```

<option symbol> contains
{ select if (<Boolean simple expression> )
  {
    <system type diagram>
    | <system type reference>
    | <block type diagram>
    | <block type reference>
    | <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 type diagram>
    | <process type reference>
    | <process area>
    | <service type diagram>
    | <service type reference>
    | <service area>
    | <procedure area>
    | <signal route definition area>
    | <create line area>
    | <option 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).

An <option area> may appear anywhere, except within <process graph area> and <process type graph area>. An <option area> may 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> or <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. Any connectors connected to an area within non-selected <option area>s are removed too.

Example

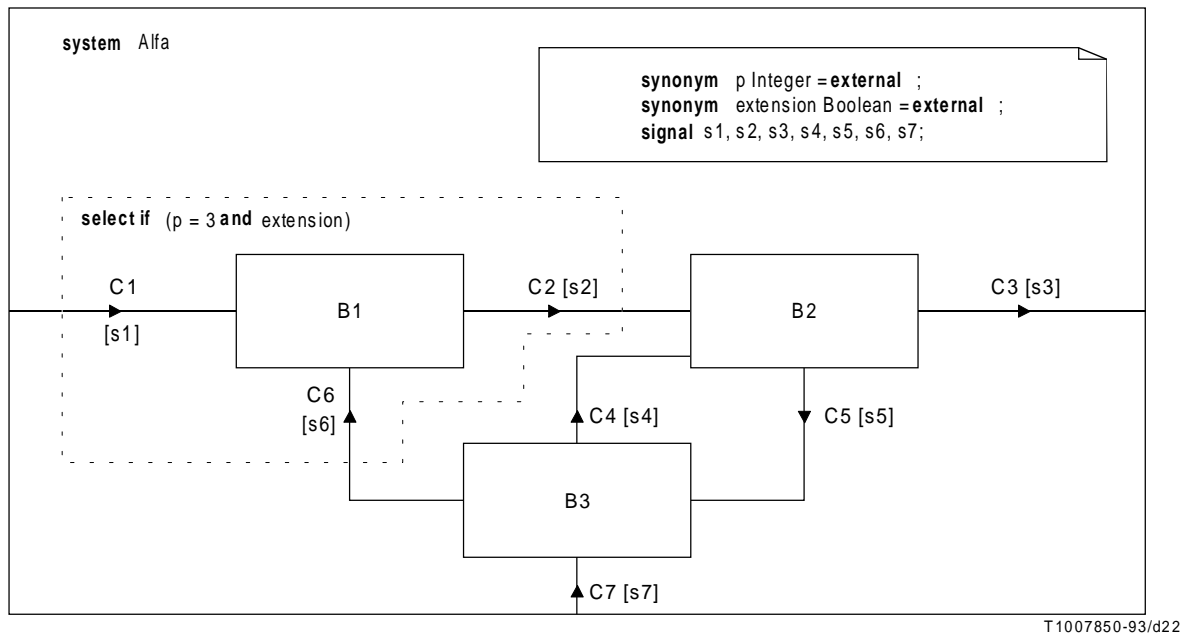
In system Alfa there are three blocks: B1, B2 and B3. Block B1 and the channels connected to it are optional, dependent on the values of the external synonyms p and extension. In SDL/PR this example is represented as follows:

```

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>.

No <answer> in <answer part>s of a <transition option> can be omitted.

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

If a <transition option> is not terminating, then it is derived syntax for a <transition option> wherein all the <answer part>s and the <else part> have inserted in their <transition>

- a) if the transition option is the last <action statement> in a <transition string>, a <join> to the following <terminator statement>, or
- b) else a <join> to the first <action statement> following the transition option.

Terminating <transition option> is defined in 2.7.5.

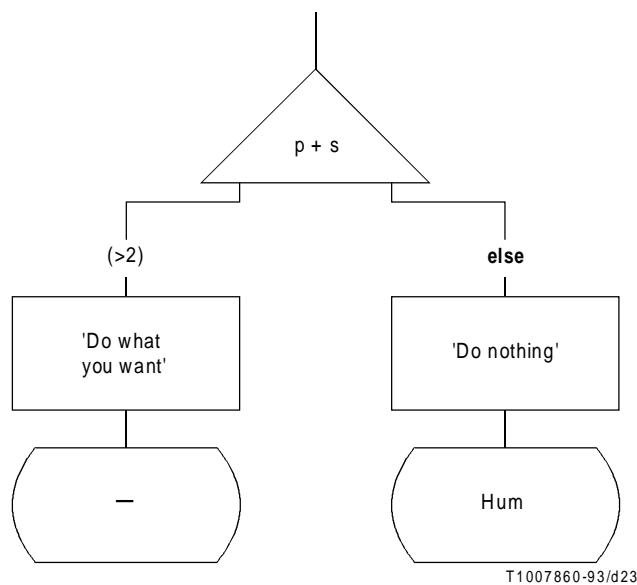
The <transition option> and <transition option area> are deleted at transformation and replaced by the contained selected constructs.

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> ::=

*

The <state name>s in an <asterisk state list> must be distinct and must be contained in other <state list>s in the enclosing <body> or in the <body> of a supertype.

Model

A <state> with an <asterisk state list> is transformed to a list of <state>s, one for each <state name> of the <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 <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 <input part>s one for each member of the complete valid input signal set of the enclosing <process definition> or <service definition>, except for <signal identifier>s of implicit input signals introduced by the additional concepts in 4.10 to 4.14 and for <signal identifier>s contained in the other <input list>s and <save list>s of the <state>.

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 input signals introduced by the additional concepts in 4.10 to 4.14 and for <signal identifier>s contained in the other <input list>s and <save list>s of the <state>.

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 Priority Input

In some cases it is convenient to express that reception of a signal takes priority over reception of other signals. This can be expressed by means of priority input.

Concrete textual grammar

<priority input> ::=

priority input [<virtuality>]

<priority input list> <end> <transition>

<priority input list> ::=

<stimulus> { , <stimulus> } *

Concrete textual grammar

<priority input association area> ::=

<solid association symbol> **is connected to** <priority input area>

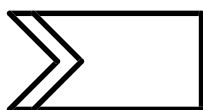
<priority input area> ::=

<priority input symbol> **contains**

{ [<virtuality>] <priority input list> }

is followed by <transition area>

<priority input symbol> ::=



Model

A <priority input> or <priority input area> which contains <virtuality> is called a virtual priority input. Virtual priority input is further described in 6.3.3.

The priority input is transformed as follows.

A state with the name `state_name` containing `<priority input>s` is split into two states. The transformation requires two implicit variables, `n` and `newn`. The variable `n` is initialized to 0. Furthermore an implicit signal `X_cont` conveying an Integer value is required. A unique `X_cont` is required for each service, if the process contains services. Priority inputs to the original state are connected to the first state, while all other inputs are connected to the second state (State2) and saved in the first state. Spontaneous transitions, remote procedure inputs and remote procedure saves of the original state are connected to both states. The transition string leading to the original state is now leading to the first state (State1). To the transition string is added the following action string:

- 1) all `<nextstate>s` which mention the `state_name` are replaced by **join l**;
- 2) The following transition is inserted:
l: task n := n+1;
output X_cont(n) to self;
nextstate State1;
- 3) To the first state is added:
input X_cont(newn);
decision (newn = n);
 (True): **nextstate State2;**
 (False): **nextstate - ;**
enddecision;

4.11 Continuous signal

In describing systems, 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 [ <virtuality> ]
    <Boolean expression> <end>
    [priority <Integer literal name> <end> ] <transition>
```

Concrete graphical grammar

```
<continuous signal association area> ::=
    <solid association symbol> is connected to <continuous signal area>

<continuous signal area> ::=
    <enabling condition symbol>
    contains { [ <virtuality> ] <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 the `<Integer literal name>`. If more `<continuous signal>s` have the same priority, a non-deterministic choice is made between these. If the value is False for all `<continuous signal>s` with an

explicit priority, the <Boolean expression>s of <continuous signal>s without priority are considered in an arbitrary order.

Model

A <continuous signal> or <continuous signal area> which contains <virtuality> is called a virtual continuous signal. Virtual continuous transition is further described in 6.3.3.

The state with the name `state_name` containing <continuous signal>s is transformed as indicated below. This transformation requires two implicit variables `n` and `newn`. The variable `n` is initialized to 0. Furthermore, an implicit signal `sx.emptyQ` conveying an Integer value is required. Here, `sx` denotes a service, if any, containing the state.

- 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 sx.emptyQ(n) to self;  
nextstate state_name;
```

- 3) The following <input part> is added to the <state> `state_name`:

```
input sx.emptyQ (newn);  
  
and a <decision> containing the <question>  
  
(newn=n)
```

- 4a) The False <answer part> contains

```
nextstate 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>). If more <continuous signal>s have the same priority they are all evaluated in an arbitrary order before the next priority level. When all <continuous signal>s with explicit priority have been evaluated, the <continuous signal>s without priority are evaluated in an arbitrary order.

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>.

The arbitrary order of evaluating a number of <continuous signal>s can be obtained by using the non-deterministic decision shorthand, i.e. a non-deterministic choice between evaluation of the possible <continuous signal>s is made, and if a <continuous signal> is evaluated to False, it is marked so that it will not be evaluated in case of a new non-deterministic choice of it. It is recorded for each round, whether more <continuous signal>s remain to be evaluated.

Example

See 4.12.

4.12 Enabling condition

In SDL, the reception of a signal or a spontaneous transition in a state immediately initiates a transition. The concept of Enabling condition makes it possible to impose an additional condition for the initiation of a transition.

Concrete textual grammar

<enabling condition> ::=

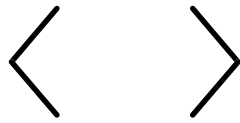
provided <Boolean expression> <end>

Concrete graphical grammar

<enabling condition area> ::=

<enabling 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 an arbitrary order before entering the state. The transformation model guarantees repeated evaluation 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. A <spontaneous transition> preceded by an <enabling condition> can be activated only if the value of the corresponding <Boolean expression> is True.

Model

The state with the name `state_name` containing <enabling condition>s is transformed as indicated below. This transformation requires two implicit variables `n` and `newn`. The variable `n` is initialized to 0. Furthermore an implicit signal `sx.emptyQ` conveying an Integer value is required. Here, `sx` denotes a service, if any, containing the state.

- 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 `sx.emptyQ (n) to self`;

A number of decisions, each containing only one <Boolean expression> corresponding to some <enabling condition> attached to the state, are added hierarchically in an arbitrary 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 distinct new state.

- 3) Each of these new states has a set of <input part>s consisting of a copy of the <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 and a set of <spontaneous transition>s whose <enabling condition>'s <Boolean expression>s evaluated to True for this state.

The <stimulus>s and the <remote procedure identifier list>s for the remaining <input part>s constitute the <save list> and <remote procedure save> for a new <save part> attached to this state. The <save part>s and <spontaneous transition>s of the original state are also copied to this new state.

- 4) Add to each of the new states:

input sx.emptyQ (newn);

A <decision> containing the <question>

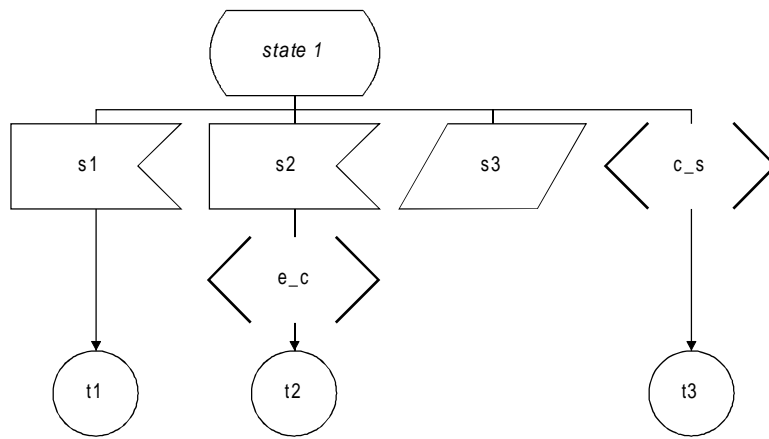
(newn=n);

- 5a) The False <answer part> contains a <nextstate> back to this same new state.
- 5b) 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 5b of the model for <enabling condition> with step 4b of the model for <continuous signal>.

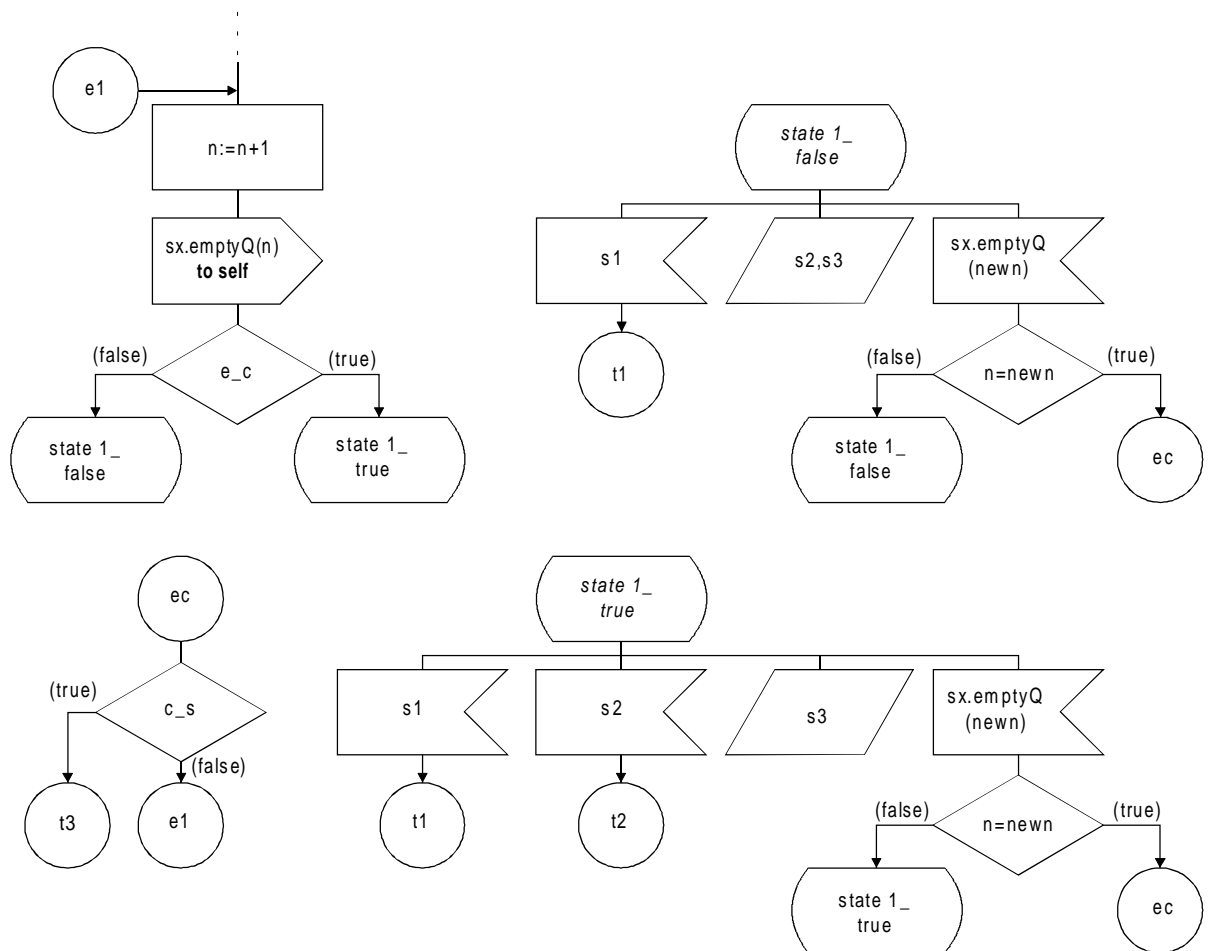
Example

An example illustrating the transformation of continuous signal and enabling condition appearing in a state is given in Figure 4.12.1.

Note in the example that the connector ec has been introduced for convenience. It is not part of the transformation model.



is transformed into



T1007870-93/d24

FIGURE 4.12.1/Z.100

Transformation of continuous 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 revealed value (see 2.6.1.1) 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 revealed values.

Concrete textual grammar

<remote variable definition> ::=

remote

<remote variable name> {,<remote variable name>}* <sort> [**nodelay**]
 {,<remote variable name> {,<remote variable name>}* <sort> [**nodelay**]}*
 <end>

<imported variable specification> ::=

imported

<remote variable identifier> {,<remote variable identifier>}* <sort>
 {,<remote variable identifier> {,<remote variable identifier>}* <sort>}* <end>

<import expression> ::=

import (<remote variable identifier> [, <destination>])

<export> ::=

export (<variable identifier> {,<variable identifier>}*)

nodelay denotes non-delaying implicit channels for the signal interchange resulting from the transformation described in *Model*.

The identifier of an <imported variable specification> must denote a possibly implicit (see below) <remote variable definition> of the same sort.

The <remote variable identifier> following **as** in an exported variable definition must denote a <remote variable definition> of the same sort as the exported variable definition. In case of no **as** clause the remote variable definition in the nearest enclosing scope unit with the same name and sort as the exported variable definition is denoted.

For each <import expression> there must exist an <imported variable specification> of the remote variable identifier in an enclosing process type, process definition, service type or service definition.

For each imported variable in a process there must exist at least one exporting process.

Remote variable definitions may be omitted and in that case, there is an implicit remote variable definition for each exported variable. The remote variable definition has the same name and sort as the exported variable and it is inserted in the scope unit where the sort is defined. The implicit remote variable definition does not have the **nodelay** property.

The <variable identifier> in <export> must denote a variable defined with **exported**.

Concrete graphical grammar

<export area> ::=

<task symbol> *contains* <export>

Semantics

A <remote variable definition> introduces the name and sort for imported and exported variables.

An exported variable definition is a variable definition with the keyword **exported**.

The association between an <imported variable specification> and an <exported variable definition> is established by both referring to the same <remote variable definition>.

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 <imported variable specification> in an importer there is a set of implicit variables, all having a unique name and the sort given in the <imported variable specification>. 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 <destination> in the <import expression>. If no <destination> is specified then the import is from an arbitrary process instance exporting the same remote variable. The association between the exported variable in the exporter and the implicit variable in the importer is specified by both referring to the same remote variable in the export variable definition and in the <imported variable specification>.

Model

An import operation is modelled 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 initialization is attached to the export variable or if the export variable is initialized when it is defined, then the implicit copy is also initialized with the same value as the export variable.

There are two implicit <signal definition>s for each <remote variable definition> in a system definition. The <signal name>s in these <signal definition>s are denoted by *xQUERY* and *xREPLY* respectively, where *x* denotes the <name> of the <remote variable definition>. The signals are defined in the same scope unit as the <remote variable definition>. The implicit copy of the exported variable is denoted by *imcx*.

a) *Importer*

The <import expression> '**import** (x, destination)' is transformed to the following, where the **to** clause is omitted, if destination is not given:

```
output xQUERY to destination;  
Wait in state xWAIT, saving all other signals;  
input xREPLY (x);
```

In all other states, xREPLY is saved.

Replace the <import expression> by x, (the <name> of the implicit variable);

b) *Exporter*

To all <state>s of the exporter, excluding implicit states derived from import, the following <input part> is added:

```
input xQUERY;  
output xREPLY (imcx) to sender/* next state the same */
```

The <export> '**export** (x)' is transformed to the following:

```
task imcx := x,
```

NOTES

1 There is a possibility of deadlock using the import construct, especially if no <destination> is given, or if <destination> does not denote a <PId expression> of a process which is guaranteed by the specification to exist at the time of receiving the xQUERY signal.

2 The optionality of remote variable definitions is introduced for compatibility with SDL-88, and is not recommended for new SDL descriptions.

4.14 Remote procedures

A client process may call a procedure defined in another process, by a request to the server process through a remote procedure call of a procedure in the server process.

Concrete textual grammar

<remote procedure definition> ::=

```
remote procedure <remote procedure name> [nodelay] <end>  
[ <procedure signature> <end> ]
```

<imported procedure specification> ::=

```
imported procedure <remote procedure identifier> <end>  
[<procedure signature> <end>]
```

<remote procedure call> ::=

```
call <remote procedure call body>
```

<remote procedure call body> ::=

```
<remote procedure identifier> [<actual parameters>]  
[ to <destination> ]
```

<remote procedure input transition> ::=

```
input [ <virtuality> ] procedure  
<remote procedure identifier list> <end>  
[<enabling condition>]  
<transition>
```

<remote procedure save> ::=

```
save [ <virtuality> ] procedure  
<remote procedure identifier list> <end>
```

<remote procedure identifier list> ::=

```
<remote procedure identifier> {,<remote procedure identifier>}*
```

nodelay denotes non-delaying implicit channels for the signal interchange resulting from the transformation described in *Model*.

The signature of <imported procedure specification> must be the same signature as the signature in the <remote procedure definition> identified by <remote procedure identifier>.

The <remote procedure identifier> following **as** in an exported procedure definition must denote a <remote procedure definition> with the same signature as the exported procedure. In an exported procedure definition with no **as** clause, the name of the exported procedure is implied and the <remote procedure definition> in the nearest surrounding scope with same name is implied.

For each <remote procedure call> there must exist an <imported procedure specification> of the remote procedure identifier in an enclosing process type, process definition, service type or service definition having the same <remote procedure name>.

For each imported procedure specification in a process there must exist at least one exported procedure in some process.

Concrete graphical grammar

<remote procedure call area> ::=

<procedure call symbol> **contains** <remote procedure call body>

<remote procedure input area> ::=

<input symbol> **contains**

{ [<virtuality>] **procedure** <remote procedure identifier list> }

is followed by { [<enabling condition area>] <transition area> }

<remote procedure save area> ::=

<save symbol> **contains**

{ [<virtuality>] **procedure** <remote procedure identifier list> <end> }

Semantics

A <remote procedure definition> introduces the name and signature for imported and exported procedures.

An exported procedure is a procedure with the keyword **exported**.

The association between an <imported procedure specification> and an exported procedure is established by both referring to the same <remote procedure definition>.

A remote procedure call by a requesting process causes the requesting process to wait until the server process has executed the procedure. Signals sent to the requesting process while waiting are saved. The server process will execute the requested procedure in the next state where save of the procedure is not specified, subject to the normal ordering of reception of signals. If neither <remote procedure save> nor <remote procedure input transition> is specified for a state, an implicit transition consisting of the procedure call only and leading back to the same state is added. If a <remote procedure input transition> is specified for a state, an implicit transition consisting of the procedure call followed by <transition> is added. If a <remote procedure save> is specified for a state, an implicit save of the signal for the requested procedure is added.

Model

Remote procedure call is modelled by an exchange of signals. These signals are implicit and are conveyed on implicit channels and signal routes. The requesting process sends a signal containing the actual parameters of the procedure call, to the server process and waits for the reply. In response to this signal, the server process executes the corresponding remote procedure, sends a signal back to the requesting process with the value of all **in/out** -parameters, and then executes the transition.

There are two implicit <signal definition>s for each <remote procedure definition>s in a <system definition>. The <signal name>s in these <signal definition>s are denoted by *pCALL* and *pREPLY* respectively, where *p* is uniquely determined. The signals are defined in the same scope unit as the <remote procedure definition>.

a) Requesting process

The <remote procedure call> '**call** Proc(apar) **to** destination' where apar is the list of actual parameters, is transformed to the following, where the **to** clause is omitted, if destination is not given:

output pCALL(apar) **to** destination;

Wait in state pWAITe, saving all other signals;

input pREPLY (aINOUTpar);

where aINOUTpar is the modified list of actual **in/out** -parameters.

In all other states, pREPLY is saved.

b) Server process

There is a declaration of an implicit variable, called ivar of sort PId.

To all <state>s with a remote procedure input transition the following <input part> is added:

input pCALL(fpar);

ivar := **sender**

call Proc(fpar);

output pREPLY (fINOUTpar) **to** ivar;

<transition>

where fpar and fINOUTpar are analogous to apar and aINOUT above.

To all <state>s, with a remote procedure save, the following <save part> is added:

save pCALL;

To all other <state>s excluding implicit states derived from input, the following <input part> is added:

input pCALL(fpar);

ivar := **sender**

call Proc(fpar);

output pREPLY (fINOUTpar) **to** ivar;

/* next state the same */

A <remote procedure input transition> or <remote procedure input area> which contains <virtuality> is called a virtual remote procedure input transition. A <remote procedure save> or <remote procedure save area> which contains <virtuality> is called a virtual remote procedure save. Virtual remote procedure input transition and save are further described in 6.3.3.

NOTE – There is a possibility of deadlock using the remote procedure construct, especially if no <destination> is given, or if <destination> does not denote a <PId expression> of a process which is guaranteed by the specification to exist at the time of receiving the pCALL signal.

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 clause 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 implementor 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 Annex D 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 defines 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 term in the language containing only literals and operators. Literals are a special case of operators without arguments.

The sorts and operators, together with the behaviour 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 data 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 a sort.

A generator is an incomplete **newtype** description: before it assumes the status of a sort, it must be transformed into a partial type definition by providing the missing information.

Some of the operators introduced by the partial type definition 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.

5.1.3 Terminology

The terminology used in clause 5 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 these 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 “term” to be equivalent to True. However, “axioms” is used to refer to a set of “axiom”s or “equation”s.

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), passive use of data (5.3) and active use of data (5.4).

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 in Annex C gives a more detailed introduction to the mathematical basis of this approach. This is formulated in a more precise mathematical way in appendix I of Annex C.

The passive use of data 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 initialization 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 having different values depending on the current values associated with variables or the current system state.

The predefined data of SDL are defined in Annex D, which defines the sorts listed in 5.1.1.

5.2 The data kernel language

The data kernel can be used to define abstract data types.

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 Annex C).

The definitions in the concrete syntax are often interdependent and cannot be separated into different scope units. For example:

```

newtype even literals 0;
  operators
    plussee: even, even  -> even;
    plusoo: odd, odd  -> even;
  axioms
    plussee(a,0)  == a;
    plussee(a,b)  == plussee(b,a);
    plusoo(a,b)  == plusoo(b,a);
endnewtype even; /* even "numbers" with plus-depends on odd */

newtype odd literals 1;
  operators
    plusoe: odd, even -> odd;
    plusoo: even, odd -> odd;
  axioms
    plusoe(a,0)  == a;
    plusoe(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 the 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 in a predefined and implicitly used package, the predefined sorts (for example Boolean and Integer) may be freely used throughout the system.

Abstract grammar

<i>Data-type-definition</i>	::	<i>Sorts</i> <i>Signature-set</i> <i>Equations</i>
<i>Sorts</i>	=	<i>Sort-name-set</i>
<i>Sort-name</i>	=	<i>Name</i>
<i>Equations</i>	=	<i>Equation-set</i>

A *data type definition* must not add new values to any *sort* of the *data type* of the enclosing <scope unit kind>.

If one *term* (see 5.2.3) is non-equivalent to another *term* according to the data type which applies in the surrounding <scope unit kind>, then these *terms* must not be defined to be equivalent by the *data type definition*.

Concrete textual grammar

```

<partial type definition> ::=
    newtype <sort name>
        [<formal context parameters>]
        [ <extended properties> ] <properties expression>
    endnewtype [ <sort name> ]

```

```

<properties expression> ::=
    <operators>
    {      <internal properties>
      |    <external properties> }
    [ <default initialization>]

```

```

<internal properties> ::=
    [ <operator definitions> ]
    [ axioms <axioms> ] [ <literal mapping> ]

```

A <formal context parameter> of <formal context parameters> must be either a <sort context parameter> or a <synonym context parameter>.

<formal context parameters>, <extended properties>, <operator definitions>, <external properties>, <literal mapping> and <default initialization> are not part of the data kernel.

The *data type definition* is represented by the collection of all the <partial type definition>s in the current <scope unit kind> combined with the *data type definition* of the surrounding <scope unit kind>.

The following <scope unit kind>s each represent an item in the abstract syntax which contains a *data type definition* : <system definition>, <block definition>, <process definition>, <service definition>, <procedure definition>, <channel substructure definition> or <block substructure definition> or the corresponding diagrams in graphical syntax. The <partial type definition> in a <system type definition>, <block type definition>, <process type definition> or <service type definition> is transformed to a <partial type definition> in the resulting (set of) instance(s), when the type is instantiated.

The *sorts* for a <scope unit kind> are represented by the set of <sort name>s introduced by the set of <partial type definition>s of the <scope unit kind>.

The *signature* set and *equations* for a <scope unit kind> are represented by the <properties expression>s of the <partial type definition>s of the <scope unit kind>.

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 kind>.

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 kind>.

The predefined data sorts have their implicit <partial type definition>s in the package Predefined as defined in Annex D.

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. 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.

Except within a <partial type definition> or a <signal refinement>, 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.

The set of sorts of a data type is the union of the set of sorts introduced in the current scope unit and the set of sorts of the data type which applies in the surrounding <scope unit kind>. The set of operators of a data type is the union of the set of operators introduced in the current scope unit and the set of operators of the data type which applies in the surrounding <scope unit kind>. The set of equations of a data type is the union of the set of equations introduced in the current scope unit and the set of equations of the data type which applies in the surrounding <scope unit kind>.

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.

5.2.2 Literals and parameterised operators

Abstract grammar

<i>Signature</i>	=	<i>Literal-signature</i> <i>Operator-signature</i>
<i>Literal-signature</i>	::	<i>Literal-operator-name</i> <i>Result</i>
<i>Operator-signature</i>	::	<i>Operator-name</i> <i>Argument-list</i> <i>Result</i>
<i>Argument-list</i>	=	<i>Sort-reference-identifier</i> ⁺
<i>Result</i>	=	<i>Sort-reference-identifier</i>
<i>Sort-reference-identifier</i>	=	<i>Sort-identifier</i> <i>Syntype-identifier</i>
<i>Literal-operator-name</i>	=	<i>Name</i>
<i>Operator-name</i>	=	<i>Name</i>
<i>Sort-identifier</i>	=	<i>Identifier</i>

Syntypes and *syntype identifiers* are not part of the kernel (see 5.3.1.9).

Concrete textual grammar

<operators> ::=	[<literal list>] [<operator list>]
<literal list> ::=	literals <literal signature> { , <literal signature> } * [<end>]
<literal signature> ::=	< <u>literal</u> <u>operator</u> name> <extended literal name>
<operator list> ::=	operators <operator signature> { <end> <operator signature> } * [<end>]
<operator signature> ::=	<operator name> : <argument list> -> <result> <ordering> <noequality>
<operator name> ::=	< <u>operator</u> 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>, <syntype> and <noequality> are not part of the data kernel.

Literals are introduced by <literal signature>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 – Guideline: an <operator signature> should mention the sort introduced by the enclosing <partial type definition> as either an <argument sort> 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

<i>Equation</i>	=	<i>Unquantified-equation</i> <i>Quantified-equations</i> <i>Conditional-equation</i> <i>Informal-text</i>
<i>Unquantified-equation</i>	::	<i>Term Term</i>
<i>Quantified-equations</i>	::	<i>Value-name-set</i> <i>Sort-identifier</i> <i>Equations</i>
<i>Value-name</i>	=	<i>Name</i>
<i>Term</i>	=	<i>Ground-term</i> <i>Composite-term</i> <i>Error-term</i>
<i>Composite-term</i>	::	<i>Value-identifier</i> <i>Operator-identifier Term⁺</i> <i>Conditional-composite-term</i>
<i>Value-identifier</i>	=	<i>Identifier</i>
<i>Operator-identifier</i>	=	<i>Identifier</i>
<i>Ground-term</i>	::	<i>Literal-operator-identifier</i> <i>Operator-identifier Ground-term⁺</i> <i>Conditional-ground-term</i>
<i>Literal-operator-identifier</i>	=	<i>Identifier</i>

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 5.3.1.6). The alternative *Error-term* in the rule *Term* is not part of the data kernel.

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.

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 5.3.1); otherwise
- b) a *value identifier* if there is a definition of that name in a <quantification> of <quantified equations> enclosing the <term>, which then must have a suitable sort for the context; otherwise
- c) a *literal operator identifier* if there is a visible literal with that name, which then must have 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 within one <unquantified equation>, or in case the <unquantified equation> is contained in a <conditional equation> then within the <conditional equation>, imply 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 is 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>, or in case the <unquantified equation> is contained in a <conditional equation> then within the <conditional 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 – Guideline: 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 $X \dots$)

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 terms denote the same value, each term denotes a different value.

Example 1

for all b in logical (eq(b,b)==T)

Example 2

$$\begin{aligned} \text{neq}(\text{T}, \text{F}) &== \text{T}; \text{neq}(\text{T}, \text{T}) == \text{F}; \\ \text{neq}(\text{F}, \text{T}) &== \text{T}; \text{neq}(\text{F}, \text{F}) == \text{F}; \end{aligned}$$

Example 3

$$\begin{aligned} \text{eq}(\mathbf{b}, \mathbf{b}) &= \mathbf{T}; \\ \text{eq}(\mathbf{F}, \text{eq}(\mathbf{T}, \mathbf{F})) &= \mathbf{T}; \\ \text{eq}(\text{eq}(\mathbf{b}, \mathbf{a}), \text{eq}(\mathbf{a}, \mathbf{b})) &= \mathbf{T}; \end{aligned}$$

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

<i>Conditional-equation</i>	::	<i>Restriction-set</i>
		<i>Restricted-equation</i>

$$\textit{Restriction} = \textit{Unquantified-equation}$$
$$\text{Restricted-equation} = \text{Unquantified-equation}$$

Concrete textual grammar

$$\langle \text{conditional equation} \rangle ::= \langle \text{restriction} \rangle \{ , \langle \text{restriction} \rangle \}^* \implies \langle \text{restricted equation} \rangle$$
$$\langle \text{restricted equation} \rangle ::= \langle \text{unquantified equation} \rangle$$
$$\langle \text{restriction} \rangle ::= \langle \text{unquantified equation} \rangle$$

Semantics

A conditional equation defines that terms denote the same value only when any value identifier in the restricted equation denotes a value which can be shown from other equations to satisfy the restrictions.

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 be called a provable conditional equation if all the restrictions (in ground terms only) can be proved to hold from unquantified equations which are not restricted equations. 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 Passive use of SDL data

Extensions to the data definition constructs in 5.2 are defined in 5.3.1. How to interpret the use of the abstract data types in expressions is defined in 5.3.3 if the expression is “passive” (that is does not depend on the system state). How to interpret expressions which are not passive (that is “active” expressions) is defined in 5.4.

5.3.1 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 5.3.1 and all subsections of 5.3.1 the relevant abstract syntax is usually to be found in 5.2.

Concrete textual grammar

<extended properties> ::=

	<inheritance rule>
	<generator transformations>
	<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>

Alternatives with <generator formal name>s are only valid in a <properties expression> in a <generator text> (see 5.3.1.12) which has that name defined as a formal parameter.

Alternatives with <exclamation> are not valid in <operator definitions>.

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 5.3.1.12).

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 5.3.1.12).

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 5.3.1.12).

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 and in <inheritance list>s.

5.3.1.1 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 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 5.3.3.1.

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 ;

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 – The equal operator (=) should not be confused with the term equivalence symbol (==).

5.3.1.2 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.

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 Annex D). They also have a special relationship with name class literals (see 5.3.1.14) and literal mappings (see 5.3.1.15). 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> pair) from the <character string>; and
- c) has a substring defined as a literal such that

substring // deleted_character_in_apostrophes

is a valid term with the same sort as the <character string literal>, where the concatenation operator and its arguments are qualified with the enclosing sort

has an implied equation given by the concrete syntax that the <character string literal> is equivalent to the substring followed by the same "/" 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', 'C', ' ';
operators "//": s, s -> s;
```

have implied equations

```
'ABC' == 'AB' // 'C' ;
'AB"' == 'AB' // " " ;
'AB'  == 'A' // 'B' ;
```

5.3.1.3 Predefined data

The predefined data including the Boolean sort which defines properties for two literals True and False, are defined in Annex D. The semantics of Equality (5.3.1.4), Boolean axioms (5.3.1.5), Conditional terms (5.3.1.6), Ordering (5.3.1.8), and Syntypes (5.3.1.9) rely on the definition of the Boolean sort (Annex D, D.1). The semantics of Name Class Literals (if <regular interval>s are used, 5.3.1.14) and Literal Mapping (5.3.1.15) also rely on the definition of Character (Annex D, D.2) and Charstring (Annex D, D.4) respectively.

In addition, the two Boolean *terms* True and False must not be (directly or indirectly) defined to be equivalent. 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.

It is also not allowed to reduce the number of values for the predefined sort PId.

Predefined data is defined in an implicitly used package Predefined (see 2.4.1.2). This package is defined in Annex D.

5.3.1.4 Equality and noequality

Concrete textual grammar

<noequality> ::=

noequality

Semantics

The symbols = and /= in the concrete syntax represent the names of the operators which are called the equal and not equal operators.

Model

Any <partial type definition> introducing some sort named S without <inheritance rule> has an implied *operator signature* pair equivalent to

"=" : S, S -> << **package** Predefined >> Boolean;

"/=" : S, S -> << **package** Predefined >> Boolean;

where Boolean is the predefined Boolean sort.

Any <partial type definition> introducing a sort named S, unless

- a) it has the keyword **noequality** in its <operator list>; or
- b) it is based on inheritance without the keyword **noequality** in its <inheritance list>

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;)
```

and an implied <literal equation>:

```
for all L1,L2 in S literals (  
  Spelling(L1) /= Spelling(L2) ==> L1 = L2 == False;)
```

5.3.1.5 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

terme <Boolean term>;

is derived syntax for the concrete syntax equation

<Boolean term> == << **package** Predefined/**type** Boolean >> True;

which has the normal relationship of an equation with the abstract syntax.

5.3.1.6 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

An equation containing a conditional term is semantically equivalent to a set of equations where all the quantified value identifiers in the Boolean term have been eliminated. This set of 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.

5.3.1.7 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 an 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.

5.3.1.8 Ordering

Concrete textual grammar

<ordering> ::=

ordering

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 -> package Predefined Boolean;  
">" : S,S -> package Predefined Boolean;  
"<=" : S,S -> package Predefined Boolean;  
">=" : S,S -> package Predefined Boolean;
```

where Boolean is the predefined Boolean sort, and also implies the Boolean *axioms*:

```
for all a, b, c, d in S  
( a = b      ==> a < b == False;  
a < b        == b > a;  
a <= b       == a < b or a = b;  
a >= b       == a > b or a = b;  
a < b == True ==> b < a == False;  
  
a < b and b = c and c < d == True ==> a < d == True);
```

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;
```

implicate A<B, B<C.

5.3.1.9 Syntypes

A syntype specifies a 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

<i>Syntype-identifier</i>	=	<i>Identifier</i>
<i>Syntype-definition</i>	::	<i>Syntype-name</i> <i>Parent-sort-identifier</i> <i>Range-condition</i>
<i>Syntype-name</i>	=	<i>Name</i>
<i>Parent-sort-identifier</i>	=	<i>Sort-identifier</i>

Concrete textual grammar

<syntype> ::=

<syntype identifier>

<syntype definition> ::=

syntype

<syntype name> = <parent sort identifier>

[<default initialization>] [**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>.

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.

When a <syntype identifier> is used as an <argument sort> 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.

If the keyword **syntype** is used and the <range condition> is omitted, then all the values of the sort are in the range condition.

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 5.4.3);
- b) output of a signal if one of the sorts specified for the signal is a syntype (see 2.5.4 and 2.7.4);
- c) calling a procedure when one of the sorts specified for the procedure **in** parameter variables is a syntype (see 2.7.3 and 2.4.6);
- 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 5.3.3.2 and 5.4.2.4);
- g) set or reset statement or active expression on a timer and one of the sorts in the timer definition is a syntype (see 2.8);
- h) remote variable or remote procedure definition if one of the sorts for derivation of implicit signals is a syntype (see 4.13 and 4.14);

- i) procedure formal context parameter with an **in/out** parameter in <procedure signature> matched with an actual context parameter where the corresponding formal parameter or the **in/out** parameter in the <procedure signature> is a syntype;
- j) any value expression, where the value returned will be within the range (see 5.4.4.6).

For example, a <syntype definition> with the keyword **syntype** and "**=** <syntype identifier>" is equivalent to substituting its <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 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.

Model

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;
```

5.3.1.9.1 Range condition

Abstract grammar

<i>Range-condition</i>	::	<i>Or-operator-identifier</i> <i>Condition-item-set</i>
<i>Condition-item</i>	=	<i>Open-range</i> <i>Closed-range</i>
<i>Open-range</i>	::	<i>Operator-identifier</i> <i>Ground-expression</i>
<i>Closed-range</i>	::	<i>And-operator-identifier</i> <i>Open-range</i> <i>Open-range</i>
<i>Or-operator-identifier</i>	=	<i>Identifier</i>
<i>And-operator-identifier</i>	=	<i>Identifier</i>

Concrete textual grammar

$\langle \text{range condition} \rangle ::=$
 $\{ \langle \text{closed range} \rangle \mid \langle \text{open range} \rangle \}$
 $\{ , \{ \langle \text{closed range} \rangle \mid \langle \text{open range} \rangle \}^* \}$

$\langle \text{closed range} \rangle ::=$
 $\langle \text{constant} \rangle : \langle \text{constant} \rangle$

$\langle \text{open range} \rangle ::=$
 $\langle \text{constant} \rangle$
 $\mid \{ = \mid / = \mid < \mid > \mid \leq \mid \geq \} \langle \text{constant} \rangle$

$\langle \text{constant} \rangle ::=$
 $\langle \text{ground expression} \rangle$

The symbol "<" (" \leq ", ">", " \geq " respectively) must only be used in the concrete syntax of the $\langle \text{range condition} \rangle$ if that symbol has been defined with an $\langle \text{operator signature} \rangle$

$P, P \rightarrow \text{package Predefined Boolean};$

where P is the sort of the syntype. These symbols represent *operator identifier*.

A $\langle \text{closed range} \rangle$ must only be used if the symbol " \leq " is defined with an $\langle \text{operator signature} \rangle$

$P, P \rightarrow \text{package Predefined Boolean};$

where P is the sort of the syntype.

A $\langle \text{constant} \rangle$ in a $\langle \text{range condition} \rangle$ 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 2.6.1.1, 5.3.1.9 and the cases where syntypes have different semantics – see the sections referenced in items a) to j) in 5.3.1.9, *Semantics*). 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. For syntype range checks, 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 ($\langle \text{open range} \rangle$ or $\langle \text{closed range} \rangle$) in the $\langle \text{range condition} \rangle$ has a corresponding *open range* or *closed range* in the *condition item*.
- b) An $\langle \text{open range} \rangle$ of the form $\langle \text{constant} \rangle$ is equivalent to an $\langle \text{open range} \rangle$ of the form $= \langle \text{constant} \rangle$.
- c) For a given term, A, then
 - i) an $\langle \text{open range} \rangle$ of the form $= \langle \text{constant} \rangle$, $/ = \langle \text{constant} \rangle$, $< \langle \text{constant} \rangle$, $\leq \langle \text{constant} \rangle$, $> \langle \text{constant} \rangle$, and $\geq \langle \text{constant} \rangle$, has sub-terms in the range check of the form $A = \langle \text{constant} \rangle$, $A / = \langle \text{constant} \rangle$, $A < \langle \text{constant} \rangle$, $A \leq \langle \text{constant} \rangle$, $A > \langle \text{constant} \rangle$, and $A \geq \langle \text{constant} \rangle$ respectively.
 - ii) a $\langle \text{closed range} \rangle$ of the form $\langle \text{first constant} \rangle : \langle \text{second constant} \rangle$ has a sub-term in the range check of the form $\langle \text{first constant} \rangle \leq A$ **and** $A \leq \langle \text{second constant} \rangle$ where **and** corresponds to the predefined 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 predefined Boolean **or** of all the elements. The range check is the term formed from the predefined Boolean **or** of all the sub-terms derived from the $\langle \text{range condition} \rangle$.

If a syntype is specified without a $\langle \text{range condition} \rangle$ then the range check is True.

5.3.1.10 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.

Unless **noequality** is specified in the <operator list>, there is an implied equation which defines the equality of structure values by elementwise equality.

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;
    Make! (x1, y1, z1) = Make! (x2, y2, z2) == (x1=x2) and (y1=y2) and (z1=z2);
endnewtype s;
```

5.3.1.11 Inheritance

Concrete textual grammar

<inheritance rule> ::=

```
inherits <sort type expression>
  [ <literal renaming> ]
  [ [ operators ] { all | ( <inheritance list> ) }
  [ <end> ] ] [ adding ]
```

<inheritance list> ::=

```
<inherited operator> { , <inherited operator> }*
[ , noequality ]
```

<inherited operator> ::=

```
[ <operator name> = ] <inherited operator name>
```

<inherited operator name> ::=

```
<base type operator name>
```

<literal renaming> ::=

```
literals <literal rename list> <end>
```

<literal rename list> ::=

```
<literal rename pair> { , <literal rename pair> }*
```

<literal rename pair> ::=

```
<literal rename signature> = <base type literal rename signature>
```

<literal rename signature> ::=

```
<literal operator name>
| <character string literal>
```

If the <sort type expression> contains <actual context parameters> then the <inheritance rule> cannot contain <literal renaming> or <inheritance list> and the keyword **all** is implied if omitted. In this case, all literals, operators and axioms of the <base type> are inherited, including the equal and the not equal operators and their associated axioms (if any).

All <literal rename signature>s in a <literal rename list> must be distinct. All the <base type 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 an operator of the <base type> defined in the <partial type definition> defining the <base type>.

The operator names "=" and "/=" are implicitly contained in <inheritance list> without any renaming.

When several operators of the <base type> have the same name as the <inherited operator name>, then all of these operators are inherited.

Model

If the <sort type expression> contains <actual context parameters> then the model for specialization in 6.3 is used, otherwise the following model is used. When the below model is applied, <operator definition>s, <operator diagram>s and <default initialization> are not inherited.

One sort (S) may be based on another (base) sort (BS) by using **newtype** in combination with an inheritance rule. The sort defined using the inheritance rule is disjoint from the base type.

If the base type has literals defined, the literal names are inherited as names for literals of the sort being defined unless literal renaming has taken place for that literal. Literal renaming has taken place for a literal if the base type 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 implied type conversion operator which takes one argument of the base type sort and has a result of the new sort. The name of this operator is the name of the sort being defined concatenated with "!".

There are inherited operators as specified by **all** or the inheritance list. The name of an inherited operator

- a) is the same as the base type 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 base type; otherwise
- b) if the base type operator name is given in the inheritance list and an operator name followed by the "=" is given for the inherited operator, is renamed to this name; otherwise
- c) if the base type operator name is given in the inheritance list and an operator name followed by "=" is not given for the inherited operator, is the same name as the base type operator name.

If neither **all** nor inheritance list is specified, the inherited operators are the equal and the not equal operators only (see 5.3.1.4).

The argument sorts and result of an inherited operator are the same as those of the corresponding operator of the base type sort, except that every occurrence of the base type in the inherited operators is changed to the new sort.

For each inherited operator (except for the "=" and "/=" operators if **noequality** is specified in <inheritance list>), there is an implied equation

$$IO(t_1, \dots, t_n) == S!(BTO(v_1, \dots, v_n)); \text{ if the result is the new sort}$$

$$IO(t_1, \dots, t_n) == BTO(v_1, \dots, v_n); \text{ otherwise}$$

where IO is the Inherited Operator, BTO is the corresponding Base Type Operator, S! is the implied type conversion operator. The v_i are disjoint value identifiers. If the sort of v_i is the base type then $t_i = S!(v_i)$, otherwise $t_i = v_i$.

For each literal rename pair $il_i = bl_i$ there is an implied equation

$$il_i == S!(bl_i);$$

There is an implied literal equation

```
for all ilv in S literals(
  for all blv in BS literals(
    Spelling(ilv) /= Spelling(il1), ..., Spelling(ilv) /= Spelling(iln),
    Spelling(ilv) == Spelling(blv) ==> ilv == S!(blv);))
```

where the il_i are the literals mentioned in literal rename pairs $il_i = bl_i$.

NOTE – From point of view of the user there is no difference in the semantics in the models since all properties for sorts with context parameters are described locally. However, the following differences exist for base sorts with and without context parameters:

	context parameter	no context parameter
Operator/literal renaming allowed	no	yes
Operator definition inherited	yes	no
Match in atleast (see 6.2.9)	no	yes, but only if no operator/literal renaming

Example

```
newtype bit
  inherits Boolean
  literals 1 = True, 0 = False;
  operators ("not", "and", "or")
  adding
  operators
    Exor: bit, bit -> bit;
  axioms
    Exor(a,b) == (a and (not b)) or ((not a) and b));
endnewtype bit;
```

5.3.1.12 Generators

A generator allows a parameterized text template to be defined which is expanded by transformation before the semantics of data types are considered.

5.3.1.12.1 Generator definition

Concrete textual grammar

```
<generator definition> ::=
    generator <generator name> ( <generator parameter list> )
    <generator text>
    endgenerator [ <generator name> ]

<generator text> ::=
    [ <generator transformations> ] <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> 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 initialization>.

Semantics

A generator names a piece of text which can be used in generator transformations.

The texts of generator transformations 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 transformed. 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;

```

5.3.1.12.2 Generator transformation

Concrete textual grammar

<generator transformations> ::=
 { <generator transformation> [<end>] [**adding**] }+

<generator transformation> ::=
 <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 transformation in extended properties or in a generator text denotes transformation of the text identified by the generator identifier. A transformed text for literals, operators and axioms is formed from the generator text with

- a) the generator actual parameters substituted for the generator parameters, and
- b) the name of the generator substituted by
 - i) if the generator transformation is in a partial type definition or syntype definition, the identity of the sort of the partial type definition or syntype definition, otherwise
 - ii) in the case of generator transformation within a generator, the name of that generator.

The transformed text for literals is the text transformed from the literals in the properties expression of the generator text omitting the keyword **literals**.

The transformed text for operators is the text transformed from the operator list in the properties expression of the generator text omitting the keyword **operators**.

The transformed text for axioms is the text transformed from the axioms in the properties expression of the generator text omitting the keyword **axioms**.

When there is more than one generator transformation in the list of generator transformations, the transformed texts for literals (operators and axioms) are formed by concatenating the transformed text for the literals (operators, axioms respectively) of all the generators in the order they appear in the list.

The transformed 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 transformations will be in the order they are transformed and before any other literals.

The transformed text for operators and axioms is added to the operator list and axioms respectively of the enclosing partial type definition, syntype definition or generator definition.

When transformed text is added to a properties expression, the keywords **literals**, **operators** and **axioms** are considered to be added to create correct concrete syntax if necessary.

Model

The abstract syntax corresponding to a generator transformation is determined after transformation. The relationship is determined from the transformed text in the context where the <generator transformation> appears.

Example

```
newtype boolbag bag(Boolean)
  adding
  operators
    yesvote : boolbag -> Boolean;
  axioms
    yesvote(b) == count(True,b) > count(False,b);
endnewtype boolbag;
```

5.3.1.13 Synonyms

A synonym gives a name to a ground expression which represents one of the values of a sort.

Concrete textual grammar

```
<synonym definition> ::=
    <internal synonym definition>
  | <external synonym definition>
```

```
<internal synonym definition> ::=
    synonym <synonym definition item>
    {, <synonym definition item> }*
```

```
<synonym definition item> ::=
    <synonym name> [ <sort> ] = <ground expression>
```

The <ground expression> in the concrete syntax denotes a *ground term* in the abstract syntax as defined in 5.3.3.2.

If a <sort> is specified, the result of the <ground expression> is bound to that *sort*. The <ground expression> denotes the corresponding *ground term* in the abstract syntax.

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.

5.3.1.14 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>.

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 Annex D, D.2).

Semantics

A name class literal is an alternative way of specifying literal signatures.

Model

The equivalent set of names of a name class literal 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 by '*' 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 by '+' 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 is a <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 Annex D, D.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 sort. The characters are considered as in uppercase, and a true prefix of a word is considered less than the whole word.

5.3.1.15 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> 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.

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 Annex D, D.4).

If the <literal signature> of the <literal operator identifier> of a <spelling term> is a <literal operator name>, 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>, 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 shown 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;
```

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>.

5.3.2 Operator definitions

Operator definitions allow operators to be defined in a manner resembling value returning procedures. However, operators must not access or change the global state. They therefore only contain a single transition. The semantics of operator definitions is expressed by transforming the transition into a start transition of a procedure.

Concrete textual grammar

```
<operator definitions> ::=
    { <operator definition> | <textual operator reference> }+
```

<operator definition> ::=

operator

```
{ <operator identifier> | <operator name> } <end>
<formal parameters> <end> <operator result> <end>
{
  <data definition>
  | <variable definition>
  | <macro definition>
  | <select definition> }*
<start> { <free action> }*
```

endoperator

[{ <operator identifier> | <operator name> }] <end>

<operator result> ::=

returns [<variable name>] <extended sort>

<textual operator reference> ::=

operator <operator name>

[<formal parameters> <operator result>]

referenced <end>

An <operator definition> or <operator diagram> must not be used to define the implied equality operators "=" and "/=".

<start> of an <operator definition> must not contain <virtuality>.

For each <operator definition> or <operator diagram>, there must exist an <operator signature> in the same scope unit having the same <operator name>, positionally having the same <argument sort>s as specified in the <formal parameters> and having the same <result> as specified in <operator result>.

For each <operator signature> at most one corresponding <operator definition> or <operator diagram> can be given.

<formal parameters> and <operator result> in <textual operator reference> may be omitted if there is no other <textual operator reference> within the same sort which has the same name. In this case, the <formal parameters> and the <operator result> are derived from the <operator signature>.

<transition> or <transition area> may neither refer to any <imperative operator>s nor any <identifier>s defined outside the enclosing <operator definition> or <operator diagram> respectively, except for <synonym identifier>s, <operator identifier>s, <literal identifier>s and <sort>s.

An operator defined by an <operator definition> or <operator diagram> must not appear in an axiom, generator or <ground expression>. An <operator definition> must not appear in a generator.

Concrete graphical grammar

<operator diagram> ::=

<frame symbol> **contains**

```
{ <operator heading>
  { { <operator text area>
    | <macro diagram> }*
  <procedure start symbol> is followed by <transition area>
  { <in-connector area> } * } set }
```

<operator heading> ::=

operator

```
{ <operator identifier> | <operator name> }
<formal parameters>
<operator result>
```

<operator text area> ::=

<text symbol> **contains**
 { <data definition>
 | <variable definition>
 | <select definition> }*

As there is no graphical grammar for sort definitions, an <operator diagram> can only be used through a <textual operator reference>.

Semantics

An operator definition is a scope unit defining its own data and variables which can be manipulated inside the transition.

Variables introduced in <formal parameters> are also modifiable.

Model

An <operator definition> or <operator diagram> is transformed into a <procedure definition> or <procedure diagram> respectively as defined in clause 7.

Application in an expression of an operator defined by an <operator definition> is transformed into a <value returning procedure call> as defined in clause 7.

5.3.3 Use of data

The following subsections define how data types, sorts, literals, operators and synonyms are interpreted in expressions.

5.3.3.1 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 5.4).

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 5.3.3.2 below.

Semantics

An expression is interpreted as the value of the ground expression or active expression. If the value is **error**, then the further behaviour of the system is undefined.

The expression has the sort of the ground expression or active expression.

5.3.3.2 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> ::=
    <sub expression>
    | <value returning procedure call>

<sub expression> ::=
    <operand0>
    | <sub expression> => <operand0>

<operand0> ::=
    <operand1>
    | <operand0> { or | xor } <operand1>

<operand1> ::=
    <operand2>
    | <operand1> and <operand2>

<operand2> ::=
    <operand3>
    | <operand2> { = | /= | > | >= | < | <= | in } <operand3>

<operand3> ::=
    <operand4>
    | <operand3> { + | - | // } <operand4>

<operand4> ::=
    <operand5>
    | <operand4> { * | / | mod | rem } <operand5>

<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 5.3.1.9.1 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> of the <operator signature> is a <syntype>, then the range check for that syntype defined in 5.3.1.9.1 is applied to the result value. The value of the range check must be True.

If an <expression> contains an <extended primary> (i.e. a <synonym>, <indexed primary>, <field primary> or <structure primary>), this is replaced at the concrete syntax level as defined in 5.3.3.3, 5.3.3.4, 5.3.3.5 and 5.3.3.6 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, e.g. 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.

5.3.3.3 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>.

5.3.3.4 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

$\langle \text{indexed primary} \rangle ::=$
 $\langle \text{primary} \rangle (\langle \text{expression list} \rangle)$

Semantics

An indexed expression represents the application of an Extract! operator.

Model

A $\langle \text{primary} \rangle$ followed by a bracketed $\langle \text{expression list} \rangle$ is derived concrete syntax for the concrete syntax

$\text{Extract!}(\langle \text{primary} \rangle, \langle \text{expression list} \rangle)$

and this is then 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 5.3.3.2.

5.3.3.5 Field primary

A field primary is a shorthand syntactic notation which can be used to denote “field selection” of “structures”. However, apart from the special syntactic form, a field primary has no special properties and denotes an operator with the primary as a parameter.

Concrete textual grammar

$\langle \text{field primary} \rangle ::=$
 $\langle \text{primary} \rangle \langle \text{field selection} \rangle$

 $\langle \text{field selection} \rangle ::=$
 $\quad ! \langle \text{field name} \rangle$
 $\quad | \quad (\langle \text{field name} \rangle \{ , \langle \text{field name} \rangle \}^*)$

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

$\langle \text{primary} \rangle (\langle \text{field name} \rangle)$

is derived syntax for

$\langle \text{primary} \rangle ! (\langle \text{field name} \rangle)$

The form

$$\langle \text{primary} \rangle (\langle \underline{\text{first field name}} \rangle \{ , \langle \underline{\text{field name}} \rangle \}^*)$$

is derived syntax for

$$\langle \text{primary} \rangle ! \langle \underline{\text{first field name}} \rangle \{ ! \langle \underline{\text{field name}} \rangle \}^*$$

where the order of field names is preserved.

The form

$$\langle \text{primary} \rangle ! \langle \underline{\text{field name}} \rangle$$

is derived syntax for

$$\langle \underline{\text{field extract operator name}} \rangle (\langle \text{primary} \rangle)$$

where the field extract operator name is formed from the concatenation of the field name and "Extract!" in that order. For example,

$$s ! fl$$

is derived syntax for

$$flExtract!(s)$$

and this is then considered as a legal expression even though flExtract! is not allowed as an operator name in the concrete syntax for expressions. The abstract syntax is determined from this concrete expression according to 5.3.3.2.

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

$$s ! name$$

5.3.3.6 Structure primary

Concrete textual grammar

$$\langle \text{structure primary} \rangle ::= [\langle \text{qualifier} \rangle] (. \langle \text{expression list} \rangle .)$$

Semantics

A structure primary represents a value of a structured sort which is constructed from expressions for each field of the structure.

The form

$$(. \langle \text{expression list} \rangle .)$$

is derived concrete syntax for

$$\text{Make!}(\langle \text{expression list} \rangle)$$

where this is considered as a legal expression even though Make! is not allowed as an operator name in concrete syntax for expressions. The abstract syntax is determined from this concrete expression according to 5.3.3.1.

5.3.3.7 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).

5.4 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.

5.4.1 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>.

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, 5.3.1.9 or 5.3.1.13.

5.4.2 Accessing variables

The following defines how an expression involving variables is interpreted.

5.4.2.1 Active expressions

Abstract grammar

$$\begin{aligned} \textit{Active-expression} &= \textit{Variable-access} \mid \\ &\textit{Conditional-expression} \mid \\ &\textit{Operator-application} \mid \\ &\textit{Imperative-operator} \mid \\ &\textit{Error-term} \end{aligned}$$

Concrete textual grammar

$$\langle \text{active expression} \rangle ::= \langle \text{active expression} \rangle$$

<active primary> ::=	
	<variable access>
	<operator application>
	<conditional expression>
	<imperative operator>
	(<active expression>)
	<active extended primary>
	error

$$\langle \text{active extended primary} \rangle ::= \langle \text{active extended primary} \rangle$$
$$\langle \text{expression list} \rangle ::= \langle \text{expression} \rangle \{ , \langle \text{expression} \rangle \}^*$$

An <expression> is an <active expression> if it contains an <active primary>.

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 5.3.3.3, 5.3.3.4, 5.3.3.5 and 5.3.3.6 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.

If an expression containing **error** is interpreted, the further behaviour of the system is undefined.

Within an active expression each operator is interpreted in the order determined by traversing the concrete syntax given in 5.3.3.2 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.

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.

5.4.2.2 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 **error** if the variable is “undefined”. If the value is **error** the further behaviour of the system is undefined.

5.4.2.3 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*.

```

<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.

5.4.2.4 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 5.3.1.9.1 applied to the value of the *expression* must be True.

Concrete textual grammar

```

<operator application> ::=
    <operator identifier> ( <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 5.3.3.2.

An operator application is an active expression which has the value of the ground term equivalent to the operator application. The equivalent ground term is determined as in 5.4.2.1.

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 5.3.1.9.1 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.

5.4.3 Assignment statement

$$\begin{array}{lcl} \textit{Assignment-statement} & :: & \textit{Variable-identifier} \\ & & \textit{Expression} \end{array}$$

If the *variable* is declared with a *syntype* and the *expression* is a *ground expression*, then the range check defined in 5.3.1.9.1 applied to the *expression* must be True.

$$\langle \text{assignment statement} \rangle ::= \langle \text{variable} \rangle := \langle \text{expression} \rangle$$

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 in the equivalent concrete syntax defined in 5.4.3.1 and 5.4.3.2 below.

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 5.3.1.9.1 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.

5.4.3.1 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 5.4.3 above.

NOTE – A consequence of this model is that a value must be assigned to a complete array, before any element can be modified.

5.4.3.2 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 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 5.3.3.5.

The concrete syntax form

$\langle \text{variable} \rangle ! \langle \text{field name} \rangle := \langle \text{expression} \rangle$

is derived concrete syntax for

$\langle \text{variable} \rangle := \langle \text{field modify operator name} \rangle (\langle \text{variable} \rangle, \langle \text{expression} \rangle)$

where

- a) the same $\langle \text{variable} \rangle$ is repeated; and
- b) the $\langle \text{field modify operator name} \rangle$ 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 $\langle \text{field modify operator name} \rangle$ is not allowed as an operator name in the concrete syntax for expressions.

If there is more than one $\langle \text{field name} \rangle$ in the field selection then they are modelled as above by expanding each $\langle \text{field name} \rangle$ in turn from right to left and considering the remaining part of the $\langle \text{field variable} \rangle$ as a $\langle \text{variable} \rangle$. For example,

$\text{var ! fielda ! fieldb} := \text{expression};$

is first modelled by

$\text{var ! fielda} := \text{fieldbModify}!(\text{var ! fielda}, \text{expression});$

and then by

$\text{var} := \text{fieldaModify}!(\text{var}, \text{fieldbModify}!(\text{var ! fielda}, \text{expression}));$

The abstract syntax is determined for the $\langle \text{assignment statement} \rangle$ formed by the modelling according to 5.4.3 above.

NOTE – A consequence of this model is that a value must be assigned to a complete structure before any field can be modified.

5.4.3.3 Default initialization

A default initialization allows initialization of all variables of a specified sort with the same value, when the variables are created.

Concrete textual grammar

$\langle \text{default initialization} \rangle ::=$

default $\langle \text{ground expression} \rangle$ [$\langle \text{end} \rangle$]

A $\langle \text{partial type definition} \rangle$ or $\langle \text{syntype definition} \rangle$ must contain not more than one $\langle \text{default initialization} \rangle$.

Semantics

A default initialization is optionally added to a properties expression of a sort. A default initialization specifies that any variable declared with the sort introduced by the partial type definition or syntype definition initially gets the value of the ground expression associated.

Model

Default initialization is a shorthand for specifying an explicit initialization for the variables of the $\langle \text{sort} \rangle$, which are declared without a $\langle \text{ground expression} \rangle$.

If no $\langle \text{default initialization} \rangle$ is given in $\langle \text{syntype definition} \rangle$, then the syntype has the $\langle \text{default initialization} \rangle$ of the $\langle \text{parent sort identifier} \rangle$ provided its value is in the range.

A variable declared inside a parameterized type whose sort is a formal context parameter, does not get the default initialization of its sort.

5.4.4 Imperative operators

Imperative operators obtain values from the underlying system state.

The transformations described in the *Models* of this section are made at the same time as the expansion for import is made. A label attached to an action in which an imperative operator appears is moved to the first task inserted during the described transformation. If several imperative operators appear in an expression, the tasks are inserted in the same order as the imperative operators appear in the expression.

Abstract grammar

$$\begin{aligned} \textit{Imperative-operator} &= \textit{Now-expression} / \\ &\textit{Pid-expression} / \\ &\textit{View-expression} / \\ &\textit{Timer-active-expression} / \\ &\textit{Anyvalue-expression} \end{aligned}$$

Concrete textual grammar

```

<imperative operator> ::=
    <now expression>
    | <import expression>
    | <PId expression>
    | <view expression>
    | <timer active expression>
    | <anyvalue expression>

```

Imperative operators are expressions for accessing the system clock, the value of imported variables, the PID values associated with a process, the value of viewed variables, the status of timers or for supplying unspecified values.

5.4.4.1 Now expression

Abstract grammar

$$Now-expression \quad :: \quad ()$$

Concrete textual grammar

```
<now expression> ::= now
```

Semantics

The now expression is an expression which accesses the 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 occurrences of **now** in the same transition will give the same value is system dependent. However, it always holds that:

```
now <= now;
```

A now expression has the Time sort.

Model

The use of `<now expression>` in an expression is a shorthand for inserting a task just before the action, where the expression occurs which assigns an implicit variable the value of `<now expression>` and then uses that implicit variable

in the expression. If <now expression> occurs several times in an expression, one variable is used for each occurrence.

5.4.4.2 Import expression

Concrete textual grammar

The concrete syntax for an import expression is defined in 4.13.

Semantics

In addition to the semantics defined in 4.13, an import expression is interpreted as a variable access (see 5.4.2.2) 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.

The use of <import expression> in an expression is a shorthand for inserting a task just before the action, where the expression occurs which assigns an implicit variable the value of <import expression> and then uses that implicit variable in the expression. If <import expression> occurs several times in an expression, one variable is used for each occurrence.

5.4.4.3 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.

5.4.4.4 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 cannot modify the value associated with the variable.

Abstract grammar

View-expression :: *View-identifier*
[*Expression*]

The *expression* must be a PId expression.

Concrete textual grammar

<view expression> ::=
view (<view identifier> [, <PId expression>])

Semantics

A view expression is interpreted in the same way as a variable access (see 5.4.2.2).

A view expression has a value which are the value of the variable access and a sort which is the sort of the *View-definition*.

If an *Expression* is given, the variable accessed is the variable in the process instance within the same block identified by *Expression*. If *Expression* denotes a non-existing instance or if the process denoted by *Expression* does not contain a variable of the same name and sort, no variable access can be made.

If no *Expression* is given, the variable accessed is the variable in an arbitrary process instance within the block, which contains a revealed variable with same name and sort. If no such instance exists, no variable access can be made.

If no variable access can be made based on <view expression> the further behaviour of the system is undefined.

Model

The use of <view expression> in an expression is a shorthand for inserting a task just before the action, where the expression occurs which assigns an implicit variable the value of <view expression> and then uses that implicit variable in the expression. If <view expression> occurs several times in an expression, one variable is used for each occurrence.

5.4.4.5 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>)])

Semantics

A timer active expression is an expression of the predefined 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). Otherwise the timer active expression has the value False. The expressions are interpreted in the order given.

If a sort specified in a timer definition is a syntype, then the range check defined in 5.3.19.1 applied to the corresponding expression in <expression list> must be True, otherwise the system is in error and the further behaviour of the system is undefined.

Model

The use of <timer active expression> in an expression is a shorthand for inserting a task just before the action, where the expression occurs which assigns an implicit variable the value of <timer active expression> and then uses that implicit variable in the expression. If <timer active expression> occurs several times in an expression, one variable is used for each occurrence.

5.4.4.6 Anyvalue expression

Abstract grammar

Anyvalue-expression :: *Sort-reference-identifier*

Concrete textual grammar

<anyvalue expression> ::= **any**(<sort>)

Semantics

An *Anyvalue-expression* is an unspecified value of the sort or syntype designated by *Sort-reference-identifier*. If no value exists, the further behaviour of the system is undefined. If *Sort-reference-identifier* denotes a *Syntype-identifier*, the resulting value will be within the range of that syntype. *Anyvalue-expression* is useful for modelling behaviour, where stating a specific value would imply over-specification. From a value returned by an *Anyvalue-expression* no assumption can be derived on other values returned by *Anyvalue-expression*.

Model

The use of <anyvalue expression> in an expression is a shorthand for inserting a task just before the action, where the expression occurs which assigns an implicit variable the value of <anyvalue expression> and then uses that implicit variable in the expression. If <anyvalue expression> occurs several times in an expression, one variable is used for each occurrence.

5.4.5 Value returning procedure call

Concrete textual grammar

<value returning procedure call> ::=
 <procedure call>
 | <remote procedure call>

A <value returning procedure call> must not occur in the <Boolean expression> of a <continuous signal area>, <continuous signal>, <enabling condition area> or <enabling condition>.

The **<procedure identifier>** in a **<value returning procedure call>** must identify a procedure (or remote procedure) having at least one parameter in its **<procedure formal parameters>** and the ending formal parameter must have the **in/out** attribute. This rule applies after the transformation of **<procedure result>**.

NOTE – Normally the **<procedure identifier>** identifies a procedure with a **<procedure result>**.

Model

The use of **<value returning procedure call>** in an expression is a shorthand for inserting a procedure call action, just before the action where the expression occurs, containing the **<value returning procedure call>** where an extra **<expression>** has been appended to the **<actual parameters>** and then using that **<expression>** instead of the **<value returning procedure call>** in the action to follow.

The constructed **<expression>** consists of the identifier of a distinct new implicit variable, whose **<sort>** is the **<sort>** of the ending formal parameter for the procedure.

The transformation takes place when other imperative operators are removed from expressions (see 5.4.4).

NOTE – Transforming the value returning procedure call after **<procedure result>** implies that a procedure with **<procedure result>** can be used in **<procedure call>** and that a procedure without **<procedure result>** can be used in **<value returning procedure call>** if it fulfils the conditions stated in this section.

5.4.6 External data

Concrete textual grammar

<external properties> ::=

alternative

<external formalism name> [, <word>] <end>

<external data description>

[endalternative] [<end>]

<external formalism name> ::=

<text>

<external data description> ::=

<text>

<external formalism name> must not contain the character ";" or ",". If **<word>** is present, it denotes the sequence which terminates the **<external data description>**. If **<external data description>** does not contain the keyword **endalternative**, **<word>** may be omitted. Otherwise, **<word>** terminates the alternative data definition and **endalternative** may be omitted. The terminating **<word>** is not formally part of the SDL description.

Semantics

alternative indicates that **<external data description>** is not formally part of the SDL description. The use of **<external formalism name>** allows to relate the **<external data description>** with some external formalism. The relations to external formalisms are not part of this Recommendation.

The literals and operators used outside **<partial type definition>** are only those which are explicitly defined in **<operators>**.

If a separate Recommendation defines the mapping from a formalism denoted by **<external formalism name>** to values based on the package Predefined (see Annex D), the mapping may imply implicit literals and operators. In this case, these are considered in addition to the ones stated in **<operators>** of the **<partial type definition>**.

The semantics of a **<partial type definition>** with **<external properties>** are conceptually assumed to be given in a set of axioms unavailable to the SDL description.

Example

```
newtype application_data

    literals empty;

    operators

        anyuseExtract! : application_data      -> Boolean;

        idExtract!     : application_data      -> Integer;

        anyuseModify!  : Boolean, application_data  -> application_data;

        idModify!      : Integer, application_data  -> application_data;

        Make!          : Boolean, Integer         -> application_data;

    alternative ASN.1;

        SET {    anyuse BOOLEAN

                id  INTEGER }

    endalternative;

endnewtype application_data;
```

6 Structural Typing Concepts in SDL

This section introduces a number of language mechanisms characterized by modeling application specific phenomena by instances and application specific concepts by types. This implies that the inheritance mechanism is intended to represent concept generalization/specialization.

The language mechanisms introduced provide

- a) (pure) type definitions that may be defined anywhere in a system or in a package;
- b) typebased instance definitions that define instances or instance sets according to types;
- c) parameterized type definitions that are independent of enclosing scope by means of context parameters and may be bound to specific scopes;
- d) specialization of supertype definitions into subtype definitions, by adding properties and by redefining virtual types and transitions.

The concepts introduced in this section are additional concepts. The properties of a shorthand notation are derived from the way it is modelled 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 defined in clause 7.

6.1 Types, instances, and gates

There is a distinction between definition of instances (or set of instances) and definition of types in SDL descriptions. This section introduces (in 6.1.1) type definitions for systems, blocks, processes, and services, and (in 6.1.3) corresponding instance specifications, while clauses 2 and 5 introduce signals, procedures, timers and sorts as types. A type definition is not connected (by channels or signal routes) to any instances; instead, type definitions introduce gates (6.1.4). These are connection points on the typebased instances for channels and signal routes.

6.1.1 Type definitions

6.1.1.1 System type

Concrete textual grammar

<system type definition> ::=

```
system type {<system type name> | <system type identifier>}
    [<formal context parameters>]
    [<specialization>] <end>
    {<entity in system>}*
endsystem type [<system type name> | <system type identifier> ] <end>
```

<textual system type reference> ::=

```
system type <system type name> referenced <end>
```

A <formal context parameter> of <formal context parameters> must not be a <process context parameter>, <variable context parameter> or <timer context parameter>.

Concrete graphical grammar

<system type diagram> ::=

```
is associated with
    <frame symbol> contains
    {<system type heading>
        {
            {<system text area>}*
            {<macro diagram>}*
            <block interaction area>
            {<type in system area>}* } set }
```

<system type heading> ::=

system type {<system type name> | <system type identifier>}
[<formal context parameters>]
[<specialization>]

<type in system area> ::=

<block type diagram>
| <block type reference>
| <process type diagram>
| <process type reference>
| <service type diagram>
| <service type reference>
| <procedure diagram>
| <graphical procedure reference>

<system type reference> ::=

<system type symbol> **contains** { **system** <system type name> }

<system type symbol> ::=

<block type symbol>

Semantics

A <system type definition> defines a system type. All systems of a system type have the same properties as defined for that system type.

6.1.1.2 Block type

Concrete textual grammar

<block type definition> ::=

[<virtuality>]
block type {<block type name> | <block type identifier>}
[<formal context parameters>]
[<virtuality constraint>]
[<specialization>] <end>
{<gate definition>}*
{<entity in block>}*
[<block substructure definition>
| <textual block substructure reference>]
endblock type [<block type name> | <block type identifier>] <end>

<textual block type reference> ::=

[<virtuality>] **block type** <block type name>
referenced <end>

A <formal context parameter> of <formal context parameters> must not be a <process context parameter>, <variable context parameter> or <timer context parameter>.

Concrete graphical grammar

<block type diagram> ::=

- <frame symbol>
- contains** { <block type heading>
 { { <block text area>* { <macro diagram>*
 { <type in block area>*
 [<process interaction area>]
 [<block substructure area>] } } **set** }
- is associated with**
 { { { <gate> }* { <graphical gate constraint> }* } } **set** }

<block type heading> ::=

- [<virtuality>]
- block type** { <block type name> | <block type identifier> }
- [<formal context parameters>] [<virtuality constraint>]
- [<specialization>]

<type in block area> ::=

- | <block type diagram>
- | <block type reference>
- | <process type diagram>
- | <process type reference>
- | <service type diagram>
- | <service type reference>
- | <procedure diagram>
- | <graphical procedure reference>

<block type reference> ::=

- <block type symbol> **contains**
 { [<virtuality>] <block name> }

<block type symbol> ::=



Semantics

A <block type definition> defines a block type. All blocks of a block type have the same properties as defined for that block type.

6.1.1.3 Process type

Concrete textual grammar

<process type definition> ::=

- [<virtuality>]
- process type** { <process type name> | <process type identifier> }
- [<formal context parameters>]
- [<virtuality constraint>]
- [<specialization>] <end>
- [<formal parameters> <end>] [<valid input signal set>]
- { <gate definition>* }
- { <entity in process>* }
- [<process type body>]
- endprocess type** [<process type name> | <process type identifier>] <end>

<process type body> ::=
 <procedure body>

<textual process type reference> ::=
 [<virtuality>] **process type** <process type name>
 referenced <end>

A <formal context parameter> of <formal context parameters> must not be a <variable context parameter> or <timer context parameter>.

Concrete graphical grammar

<process type diagram> ::=
 <frame symbol>
 contains {<process type heading>
 { {<process text area>}*
 {<type in process area>}*
 {<macro diagram>}*
 {<process type graph area> | <service interaction area> } **set** }
 is associated with
 { { {<gate>}* {<graphical gate constraint>}* } **set** }

<process type heading> ::=
 [<virtuality>]
 process type {<process type name> | <process type identifier>}
 [<formal context parameters>]
 [<virtuality constraint>]
 [<specialization>] [<end>]
 [<formal parameters>]

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

<type in process area> ::=
 <service type diagram>
 | <service type reference>
 | <procedure diagram>
 | <graphical procedure reference>

<process type reference> ::=
 <process type symbol> **contains**
 { [<virtuality>] <process type name> }

<process type symbol> ::=



Semantics

A <process type definition> defines a process type. All process instance sets of a process type have the same properties as defined for that process type.

The complete valid input signal set of a process type is the union of the complete valid input signal set of its supertype, the <signal list>s in all gates in the direction towards the process type, the <valid input signal set>, the implicit input signals introduced by the additional concepts in 4.10 to 4.14 and the timer signals.

Signals mentioned in <output>s of a process type must be in the complete valid input signal set of the process type or in the <signal list> of a gate in the direction from the process type.

6.1.1.4 Service type

Concrete textual grammar

```
<service type definition> ::=
    [ <virtuality> ]
    service type { <service type name> | <service type identifier> }
        [<formal context parameters>]
        [<virtuality constraint>]
        [<specialization>] <end>
        [<valid input signal set>]
        {<gate definition>}*
        {<entity in service>}*
        [ <service type body> ]
    endservice type [ { <service type name> | <service type identifier> } ] <end>
```

```
<service type body> ::=
    <process type body>
```

```
<textual service type reference> ::=
    [<virtuality>] service type <service type name>
    referenced <end>
```

A <formal context parameter> of <formal context parameters> must not be a <timer context parameter>.

A <variable definition> in a <service type definition> must not contain the keyword **revealed**.

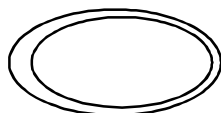
Concrete graphical grammar

```
<service type diagram> ::=
    <frame symbol> contains
    { <service type heading>
    { { <service text area> } *
      { <graphical procedure reference> } *
    { <procedure diagram> } *
    { <macro diagram> } *
    <service graph area> } set }
    is associated with
    { { { <gate> } * { <graphical gate constraint> } * } set }
```

```
<service type heading> ::=
    [<virtuality>]
    service type { <service type name> | <service type identifier> }
        [<formal context parameters>]
        [<virtuality constraint> ]
        [<specialization>]
```

```
<service type reference> ::=
    <service type symbol> contains
    { [<virtuality>] <service type name> }
```

```
<service type symbol> ::=
```



Semantics

A *<service type definition>* defines a service type. All services of a service type have the same properties as defined for that service type.

The complete valid input signal set of a service type is the union of the complete valid input signal set of its supertype, the *<signal list>*s in all gates in the direction towards the service type, the *<valid input signal set>*, the implicit signals introduced by the additional concepts in 4.10 to 4.14 and the timer signals.

Signals mentioned in *<output>*s of a service type must be in the complete valid input signal set of the service type or in the *<signal list>* of a gate in the direction from the service type.

6.1.2 Type expression

A type expression is used for defining one type in terms of another as defined by specialization in 6.3.

Concrete textual grammar

<type expression> ::=

<base type> [<actual context parameters>*]*

<base type> ::=

<identifier>

<actual context parameters> can be specified if and only if *<base type>* denotes a parameterized type. Context parameters are defined in 6.2.

Outside a parameterized type, the parameterized type can only be used by referring to its *<identifier>* in *<type expression>*.

Semantics

A *<type expression>* yields either the type identified by the identifier of *<base type>* in case of no actual context parameters or an anonymous type defined by applying the actual context parameters to the formal context parameters of the parameterized type denoted by the identifier of *<base type>*.

If some actual context parameters are omitted, the type is still parameterized.

A *<type expression>* does not represent specialization if the *<base type>* is a parameterized type (see 6.3).

NOTE – Even though the definition denoted by the *<base type>* fulfils any static conditions, usage of the *<type expression>* may violate the properties associated with the *<base type>*. The properties may be violated in the following cases:

- 1) When a scope unit has signal context parameters or timer context parameters, the condition that stimuli for a state must be disjoint, depends on which actual context parameters that will be used.
- 2) When an output in a scope unit refers to a gate, a signal route or a channel, which is not defined in the nearest enclosing type having gates, instantiation of that type result in an erroneous specification if there is no communication path to the gate.
- 3) When a procedure contains references to signal identifiers, remote variables and remote procedures, specialization of that procedure inside a process or service result in an erroneous specification if the usage of such identifiers inside the procedure violates valid usage for the process or service.
- 4) When a create action, output action or view definition inside a process or service type defined in a block refers to a process instance set, specialization and/or instantiation of the process type in a substructure of the block results in an erroneous specification.
- 5) When services types are instantiated, the resulting process is erroneous if two or more services have the same signal in the complete valid input signal set.

- 6) When a scope unit has a process context parameter which is used in an output action, the existence of a possible communication path depends on which actual context parameter will be used.
- 7) When a scope unit has a sort context parameter and an operator in the sort signature is used in axioms, application of an actual sort context parameter for which the operator is defined using an operator definition results in an erroneous specification.
- 8) If a formal parameter of a procedure added in a specialization has the <parameter kind> **in/out**, a call in the supertype to a subtype (using **this**) will result in an omitted actual **in/out** parameter, i.e. an erroneous specification.
- 9) If a formal procedure context parameter is defined with an **atleast** constraint and the actual context parameter has added a parameter of <parameter kind> **in/out**, a call of the formal procedure context parameter in the parameterized type may result in an omitted actual **in/out** parameter, i.e. an erroneous specification.

Model

If the scope unit contains <specialization> and any <actual context parameter>s are omitted in the <type expression>, the <formal context parameter>s are copied (while preserving their order) and inserted in front of the <formal context parameter>s (if any) of the scope unit. In place of omitted <actual context parameter>s, the names of corresponding <formal context parameter>s are inserted. These <actual context parameter>s now have the defining context in the current scope unit.

6.1.3 Definitions based on types

A typebased system, block, process or service definition defines a system, block, process instance set or service, respectively, according to a type denoted by <type expression>. The defined entities get the properties of the types, they are based on.

6.1.3.1 System definition based on system type

Concrete textual grammar

<textual typebased system definition> ::=

<typebased system heading> <end>

<typebased system heading> ::=

system <system name> : <system type expression>

Concrete graphical grammar

<graphical typebased system definition> ::=

<frame symbol> **contains** <typebased system heading>

Semantics

A typebased system definition defines a *System-definition* derived by transformation from a system type.

Model

A <textual typebased system definition> or a <graphical typebased system definition> is transformed to a <system definition> which has the definitions of the system type as defined by <system type expression>.

6.1.3.2 Block definition based on block type

Concrete textual grammar

<textual typebased block definition> ::=

block <typebased block heading> <end>

<typebased block heading> ::=

<block name> [<number of block instances>] : <block type expression>

<number of block instances> ::=

(<Natural simple expression>)

If <number of block instances> is omitted, the number of blocks is 1. The <number of block instances> must be equal to or greater than 1.

Concrete graphical grammar

<graphical typebased block definition> ::=

<block symbol> **contains**
{ <typebased block heading>
 { <gate>* } **set** }

<existing typebased block definition> ::=

<dashed block symbol> **contains** { <block identifier> { <gate>* } **set** }

<dashed block symbol> ::=



The <gate>s are placed near the border of the symbols and associated with the connection point to channels.

Semantics

A typebased block definition defines *Block-definitions* derived by transformation from a block type.

Model

A <textual typebased block definition> or a <graphical typebased block definition> is transformed to a <block definition> which has the definitions of the block type as defined by <block type expression>. The number of derived <block definition>s is specified by <number of block instances>.

An <existing typebased block definition> can only appear in a subtype definition. It represents the block defined in the supertype of the subtype definition. Additional channels connected to gates of the existing block may be specified.

6.1.3.3 Process definition based on process type

Concrete textual grammar

<textual typebased process definition> ::=

process <typebased process heading> <end>

<typebased process heading> ::=

<process name> [<number of process instances>] : <process type expression>

The process type denoted by <base type> in <process type expression> must contain a start transition.

Concrete graphical grammar

<graphical typebased process definition> ::=

<process symbol> **contains**
{ <typebased process heading>
 { <gate>* }**set** }

<existing typebased process definition> ::=

<dashed process symbol> **contains** { <process identifier> { <gate>* }**set** }

<dashed process symbol> ::=



The <gate>s are placed near the border of the symbols and associated with the connection point to signal routes.

Semantics

A typebased process definition defines a *Process-definition* derived by transformation from a process type.

Creation of individual process instances are described in 2.4.4 (system initialization) and 2.7.2 (dynamic create request).

Model

A <textual typebased process definition> or a <graphical typebased process definition> is transformed to a <process definition> which has the definitions of the process type as defined by <process type expression>.

An <existing typebased process definition> can only appear in a subtype definition. It represents the process defined in the supertype of the subtype definition. Additional signal routes connected to gates of the existing process may be specified.

6.1.3.4 Service definition based on service type

Concrete textual grammar

<textual typebased service definition> ::=

service <typebased service heading> <end>

<typebased service heading> ::=

<service name> : <service type expression>

The service type denoted by <base type> in <service type expression> must contain a start transition.

Concrete graphical grammar

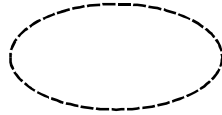
<graphical typebased service definition> ::=

<service symbol> **contains**
{ <typebased service heading>
 { <gate>* }**set** }

<existing typebased service definition> ::=

<dashed service symbol> **contains** { <service identifier> { <gate>* }**set** }

<dashed service symbol> ::=



The <gate>s are placed near the border of the symbols and associated with the connection point to signal routes.

Semantics

A typebased service definition defines a *Service-definition* derived by transformation from a service type.

Model

A <textual typebased service definition> or a <graphical typebased service definition> is transformed to a <service definition> which has the definitions of the service type as defined by <service type expression>.

An <existing typebased service definition> can only appear in subtype definitions. It represents a service specified in the supertype of the subtype definition. Additional signal routes connected to gates of the existing service may be specified.

6.1.4 Gate

Gates are defined in block, process or service types (as defined in 6.1.1) and represent connection points for channels and signal routes, connecting instances of these types (as defined in 6.1.3) with other instances of the same entity kind or with the enclosing frame symbol.

Concrete textual grammar

<gate definition> ::=

```
gate <gate>
[adding] <gate constraint> <end>
[ <gate constraint> <end> ]
```

<gate> ::=

```
<gate name>
```

<gate constraint> ::=

```
{      out [ to <textual endpoint constraint> ]
  |    in [ from<textual endpoint constraint> ] }
[ with <signal list> ]
```

<textual endpoint constraint> ::=

```
[atleast]<identifier>
```

out or **in** denotes the direction of <signal list>, from or to the type respectively. Types from which instances are defined must have a <signal list> in the <gate constraint>s.

The <identifier> of <textual endpoint constraint> must denote a type definition of the same entity kind as the type definition in which the gate is defined.

A channel/signal route connected to a gate must be compatible with the gate constraint. A channel/signal route is compatible with a gate constraint if the other endpoint of the channel/signal route is a block/process/service of the type denoted by <identifier> in the endpoint constraint or a subtype of this type (in case **atleast**<identifier>), and if the set of signals on the channel/signal route is equal to or is a subset of the set of signals specified for the gate in the respective direction.

<base type> in <textual typebased block definition> or <textual typebased process definition> must for each combination of (gate, signal, direction) defined by the type contain at least one signal route mentioning **env**, the gate and the signal for the given direction, if the base type contains signal routes.

A <block substructure definition> in <base type> in <textual typebased block definition> must for each combination of (gate, signal, direction) defined by the block type contain at least one channel mentioning **env**, the gate and the signal for the given direction.

Where two <gate constraint>s are specified one must be in the reverse direction to the other, and the <textual endpoint constraint>s of the two <gate constraint>s must be the same.

adding may only be specified in a subtype definition and only for a gate defined in the supertype. When **adding** is specified for a <gate>, any <textual endpoint constraint>s and <signal list>s are additions to the <gate constraint>s of the gate in the supertype.

If <textual endpoint constraint> is specified for the gate in the supertype, the <identifier> of an (added) <textual endpoint constraint> must denote the same type or a subtype of the type denoted in the <textual endpoint constraint> of the supertype.

Concrete graphical grammar

```
<graphical gate constraint> ::=
    { <gate symbol> | <existing gate symbol> }
    is associated with
    { <gate> [ <signal list area>[<signal list area>] ] }
    is connected to
    { <frame symbol> [ <endpoint constraint> ] }
```

```
<endpoint constraint> ::=
    { <block symbol> | <process symbol> | <service symbol> }
    contains <textual endpoint constraint>
```

```
<gate symbol> ::=
    <signal route symbol>
```

```
<existing gate symbol> ::=
    <gate symbol 1> | <gate symbol 2>
```

```
<gate symbol 1> ::=
    — — — ➔
```

```
<gate symbol 2> ::=
    ➔ — — ➔
```

The <graphical gate constraint> is outside the diagram frame.

The <signal list area> elements are associated with the directions of the signal route symbol.

The symbol in <endpoint constraint> must be the symbol for instance definition corresponding to the type definition in which the gate is defined, i.e. <block symbol>, <process symbol> or <service symbol>.

<signal list area>s and <endpoint constraint> associated with an <existing gate symbol> are regarded as additions to those of the gate definition in the supertype.

An <existing gate symbol> can only appear in a subtype definition, and it is then a representative for the gate with the same <gate name> specified in the supertype of the subtype definition.

For each arrowhead on the <gate symbol>, there can be a <signal list area>. A <signal list area> must be unambiguously close enough to the arrowhead to which it is associated. The arrowhead indicates whether the <signal list area> denotes an **in** or an **out** <gate constraint>.

Semantics

The use of gates in type definitions corresponds to the use of communication paths in the enclosing scope in (set of) instance specifications.

Model

For each instance of a type defining a <gate>, a <signal route to route connection>, a <channel to route connection> or a <channel connection> is derived:

- a) For each instantiation of a process type, a <signal route to route connection> is derived in the resulting <process definition> where:
 - The <external signal route identifiers> are those signal routes defined in the enclosing block which mention the process and the gate in a <signal route path>.
 - The <signal route identifiers> are those signal routes defined inside the <process definition> which mention the keyword **env** and the gate in the <signal route path>.
- b) For each instantiation of a block type, a <channel to route connection> in the resulting <block definition> is derived where:
 - The <channel identifiers> are those channels defined in the scope unit enclosing the block which mention the block and the gate in a <channel path>.
 - The <signal route identifiers> are those signal routes defined inside the <block definition> which mention the keyword **env** and the gate in the <channel path>.
- c) For each instantiation of a block type containing a <block substructure definition>, a <channel connection> in the resulting scope of the block is derived where:
 - The <channel identifiers> are those channels defined in the scope unit enclosing the surrounding block which mention the block and the gate in a <channel path>.
 - The <subchannel identifiers> are those channels defined inside the <block substructure definition> which mention the keyword **env** and the gate in a <channel path>. Any rules for using signal refinement on type-based block instances is defined through the rules for the resulting <channel connection> (see 3.3).

6.2 Context parameter

In order for a type definition to be used in different contexts, both within the same system specification and within different system specifications, types may be parameterized by context parameters. Context parameters are replaced by actual identifiers, actual context parameters, as defined in 6.1.2.

The following type definitions can have formal context parameters: system type, block type, process type, service type, procedure, signal and sort.

Context parameters can be given constraints, i.e. required properties any entity denoted by the corresponding actual identifier must have. The context parameters have these properties inside the type.

Concrete textual grammar

<formal context parameters> ::=

```
<context parameters start> <formal context parameter>
{<end> <formal context parameter> }* <context parameters end>
```

<actual context parameters> ::=

```
<context parameters start>
[<actual context parameter>] { , [<actual context parameter> ] }* <context parameters end>
```

```

<actual context parameter> ::=
    <identifier>

<context parameters start> ::=
    <

<context parameters end> ::=
    >

<formal context parameter> ::=
    <process context parameter>
    | <procedure context parameter>
    | <remote procedure context parameter>
    | <signal context parameter>
    | <variable context parameter>
    | <remote variable context parameter>
    | <timer context parameter>
    | <synonym context parameter>
    | <sort context parameter>

```

NOTE – The characters "<" and ">" delimit context parameters, and are not solely used as meta-symbols above.

The scope unit of a type definition with formal context parameters defines the names of the formal context parameters. These names are therefore visible in the definition of the type, and also in the definition of the formal context parameters.

Formal context parameters can neither be used as <base type> in <type expression> nor in **atleast** constraints of <formal context parameters>.

Constraints are specified by constraint specifications. A constraint specification introduces the entity of the formal context parameter followed by either a constraint signature or an **atleast** clause. A constraint signature introduces directly sufficient properties of the formal context parameter. An **atleast** clause denotes that the formal context parameter must be replaced by an actual context parameter, which is the same type or a subtype of the type identified in the **atleast** clause. Identifiers following the keyword **atleast** in this clause must identify type definitions of the entity kind of the context parameter and must be neither formal context parameters nor parameterized types.

A formal context parameter of a type must be bound only to an actual context parameter of the same entity kind which meets the constraint of the formal parameter.

The parameterized type can only use the properties of a context parameter which are given by the constraint, except for the cases listed in 6.1.2.

A context parameter using other context parameters in its constraint cannot be bound before the other parameters are bound.

The binding of an actual variable or synonym context parameter to its definition is not resolved by context.

Trailing commas may be omitted in <actual context parameters>.

Semantics

The formal context parameters of a type definition that is neither a subtype definition nor defined by binding formal context parameters in a <type expression> are the parameters specified in the <formal context parameters>.

Context parameters of a type are bound in the definition of a <type expression> or an <inheritance rule> to actual context parameters. In this binding, occurrences of formal context parameters inside the parameterized type are replaced by the actual parameters. During this binding of identifiers contained in <formal context parameter>s to definitions (i.e. deriving their qualifier, see 2.2.2), other local definitions than the <formal context parameters>s are ignored.

Parameterized types cannot be actual context parameters. In order for a definition to be allowed as an actual context parameter, it must be of the same entity kind as the formal parameter and satisfy the constraint of the formal parameter.

Model

If a scope unit contains <specialization>, any omitted actual context parameter in the <specialization> is replaced by the corresponding <formal context parameter> of the <base type> in the <type expression> and this <formal context parameter> becomes a formal context parameter of the scope unit.

6.2.1 Process context parameter

Concrete textual grammar

```

<process context parameter> ::=
    process <process name> <process constraint>

<process constraint> ::=
    [atleast] <process identifier> | <process signature>

<process signature> ::=
    [[ <end> ] <formal parameters signature> ]

<formal parameters signature> ::=
    fpar <sort>{, <sort> }*

```

Semantics

An actual process parameter must identify a process definition. Its type must be a subtype of the constraint process type (**atleast** <process identifier>) with no addition of formal parameters to those of the constraint type, or it must be the type denoted by <process identifier>, or it must be compatible with the formal process signature. A process definition is compatible with the formal process signature if the formal parameters of the process definition have the same sorts as the corresponding <sort>s of the <formal parameters signature>.

6.2.2 Procedure context parameter

Concrete textual grammar

```

<procedure context parameter> ::=
    procedure <procedure name> <procedure constraint>

<procedure constraint> ::=
    atleast <procedure identifier>
    | <procedure signature>

<procedure signature> ::=
    [[ <end> ] fpar <procedure formal parameter constraint>
        {, <procedure formal parameter constraint> }*
        [ <end> returns <sort> ]
    | [ <end> ] returns <sort>

<procedure formal parameter constraint> ::=
    <parameter kind> <sort>

```

Semantics

An actual procedure parameter must identify a procedure definition that is either a specialization of the procedure of the constraint (**atleast** <procedure> identifier) or compatible with the formal procedure signature. A procedure definition is compatible with the formal procedure signature

- a) if the formal parameters of the procedure definition have the same sorts as the corresponding parameters of the signature, if they have the same <parameter kind>, and if both return a value of the same <sort> or if neither returns a value, or
- b) if this rule is obeyed after substituting a result specification into an additional **in/out** parameter; and
- c) if each **in/out** parameter in the procedure definition has the same <sort> identifier or <syntype> identifier as the corresponding parameter of the signature.

6.2.3 Remote procedure context parameter

Concrete textual grammar

<remote procedure context parameter> ::=

remote procedure <procedure name> <procedure signature>

Semantics

An actual parameter to a **remote** procedure context parameter must identify a <remote procedure definition> with the same signature.

6.2.4 Signal context parameter

Concrete textual grammar

<signal context parameter> ::=

signal <signal name> <signal constraint>
 {, <signal name> <signal constraint> }*

<signal constraint> ::=

atleast <signal identifier>
 | <signal signature>

<signal signature> ::=

[<sort list>]
[<signal refinement>]

Semantics

An actual signal parameter must identify a signal definition that is either a subtype of the signal type of the constraint (**atleast** <signal> identifier) or compatible with the formal signal signature.

A signal definition is compatible with a formal signal signature if the sorts of the signal are the same sorts as in the sort constraint list, and if <signal refinement> of the <signal context parameter> is stated, the signal definitions of the <signal refinement> are compatible with the <signal refinement> of the <signal context parameter>.

Two <signal refinement>s are compatible, if they define the same set of signal names, and if each corresponding <subsignal definition> has the same <reverse> attribute.

6.2.5 Variable context parameter

Concrete textual grammar

```
<variable context parameter> ::=  
    dcl <variable name> {,<variable name>}* <sort>  
        {, <variable name> {,<variable name>}* <sort> }*
```

Semantics

An actual parameter must be a variable or a formal process or procedure parameter of the same sort as the sort of the constraint.

6.2.6 Remote variable context parameter

Concrete textual grammar

```
<remote variable context parameter> ::=  
    remote <remote variable name> {,<remote variable name>}* <sort>  
        {, <remote variable name> {,<remote variable name>}* <sort> }*
```

Semantics

An actual parameter must identify a <remote variable definition> of the same sort.

6.2.7 Timer context parameter

Concrete textual grammar

```
<timer context parameter> ::=  
    timer <timer name> <timer constraint>  
        {, <timer name> <timer constraint> }*
```

```
<timer constraint> ::=  
    [<sort list>]
```

Semantics

An actual timer parameter must identify a timer definition that is compatible with the formal sort constraint list. A timer definition is compatible with a formal sort constraint list if the sorts of the timer are the same sorts as in the sort constraint list.

6.2.8 Synonym context parameter

Concrete textual grammar

```
<synonym context parameter> ::=  
    synonym <synonym name> <synonym constraint>  
        {, <synonym name> <synonym constraint> }*
```

```
<synonym constraint> ::=  
    <sort>
```

Semantics

An actual synonym must be of the same sort as the sort of the constraint.

6.2.9 Sort context parameter

Concrete textual grammar

```
<sort context parameter> ::=  
    newtype <sort name> [ <sort constraint>]  
  
<sort constraint> ::=  
    atleast <sort>  
    | <sort signature>  
  
<sort signature> ::=  
    <operators> endnewtype [<sort name>]
```

Semantics

If <sort constraint> is omitted, the actual sort can be any sort. Otherwise, an actual sort must be either a subtype without <literal renaming> or <inheritance list> of the sort of the constraint (**atleast** <sort>), or compatible with the formal sort signature. A sort is compatible with the formal sort signature if the literals of the sort include the literals in the formal sort signature and the operators of the sort include the operators in the formal sort signature and the operators have the same signatures.

A <sort signature> implicitly includes the equal and the not equal operators (see 5.3.1.4). **noequality** is not allowed in <sort signature>.

ordering in <sort signature> yields the signature for the ordering operators as defined in 5.3.1.8.

6.3 Specialization

In order to express concept specialization, a type may be defined as a specialization of another type (the supertype), yielding a new subtype. A subtype may have properties in addition to the properties of the supertype, and it may redefine virtual local types and transitions.

Specialization is in the specialized type definition specified by "**inherits** <type expression>", where <type expression> denotes the general type. The general type is said to be the supertype of the specialized type, and the specialized type is said to be a subtype of the general type. Any specialization of the subtype is a subtype of the general type.

Note that the whole <type expression> represents the supertype. Only if the <type expression> contains only <base type>, the supertype is named.

Virtual types can be given constraints, i.e. properties any redefinition of the virtual type must have. These properties are used to guarantee properties of any redefinition. Virtual types are defined in 6.3.2.

6.3.1 Adding properties

Concrete textual grammar

```
<specialization> ::=  
    inherits <type expression> [adding]
```

If a type subT is derived from a (super)type T through specialization (either directly or indirectly), then

- a) T must not enclose subT;
- b) T must not be derived from subT;
- c) definitions enclosed by T must not be derived from subT.

The resulting content of a specialized type definition with local definitions consists of the content of the supertype followed by the content of the specialized definition. This implies that the set of definitions of the specialized definition is the union of those given in the specialized definition itself and those of the supertype. The resulting set of definitions must obey the rules for distinct names as given in 2.2.2. However, three exceptions to this rule are:

- 1) a redefinition of a virtual type is a definition with the same name as that of the virtual type;
- 2) a gate of the supertype may be given an extended definition (in terms of signals conveyed and endpoint constraints) in a subtype – this is specified by a gate definition with the same name as that of the supertype;
- 3) if the <type expression> contains <actual context parameters> any occurrence of the <base type> of the <type expression> is replaced by the name of the subtype.

The <block substructure definition> given in a specialized block type definition is added to the substructure definition of the block supertype. If present, the name of the substructure of the subtype must be the same as the name of the substructure of the supertype.

The formal context parameters of a subtype are the unbound, formal context parameters of the supertype definition followed by the formal context parameters of the specialized type (see 6.2).

The formal parameters of a specialized process type or procedure are the formal parameters of the process supertype or procedure followed by the formal parameters added in the specialization.

The complete valid input signal set of a specialized process or service type is the union of the complete valid input signal set of the specialized process or service type and the complete valid input signal set of the process or service supertype respectively.

The resulting graph of a specialized process type, service type or procedure definition consists of the graph of its supertype definition followed by the graph of the specialized process type, service type or procedure definition.

The process graph of a given process type, service type or procedure definition may have at most one start transition.

All the <connector name>s defined in the combined <body> must be distinct. It is permissible to have a join from the <body> of the specialized process/procedure/service to a connector defined in the supertype.

A specialized signal definition may add (by appending) sorts to the sort list of the supertype.

A specialized partial type definition may add properties in terms of operators, literals, axioms, operator definitions and default assignment.

NOTE – When a gate in a subtype is an extension of an existing gate in a supertype, the <existing gate symbol> is used in SDL/GR.

6.3.2 Virtual type

A type being defined locally to a type definition (*enclosing type*) may be specified to be a virtual type. Block types, process types, service types, and procedures may be specified as virtual types. A virtual type may be constrained by another type of the same entity kind and it may be redefined in subtypes of its enclosing type.

Concrete textual grammar

<virtuality> ::=

virtual | redefined | finalized

<virtuality constraint> ::=

atleast <identifier>

The syntax of virtual types are introduced in 6.1.1.2 (block type), 6.1.1.3 (process type), 6.1.1.4 (service type) and 2.4.6 (procedure).

A virtual type and its constraint cannot have context parameters.

A virtual type may be constrained by the type identified by the <identifier> following the keyword **atleast**. This <identifier> must identify a type definition of the same entity kind as the virtual type. If <virtuality constraint> is omitted, it corresponds to specifying the virtual type itself as the constraint.

If <virtuality> is present in both the reference and the referenced definition, then they must be equal. If <procedure preamble> is present in both procedure reference and in the referenced procedure definition they must be equal.

A virtual type must have <virtuality> in its definition.

A virtual type must have the same formal parameters, gates and signals on its gates as its constraint.

Semantics

A virtual type may be redefined in the definition of a subtype of the enclosing type of the virtual type. In the subtype it is the definition from the subtype that defines the type of instances of the virtual type, also when applying the virtual type in parts of the subtype inherited from the supertype. A virtual type that is not redefined in a subtype definition has the definition as given in the supertype definition.

Accessing a virtual type by means of a qualifier denoting one of the supertypes implies, however, the application of the (re)definition of the virtual type given in the actual supertype denoted by the qualifier. A type (T) whose name is hidden in an enclosing subtype by a redefinition (of T) can be made visible through qualification with a supertype name (i.e. a type name in the inheritance chain). The qualifier will consist of only one path item denoting the particular supertype, or in the case of accessing a hidden type from a substructure of the supertype, the qualifier will consist of two path items, the first one denoting the block type and the second one denoting the substructure.

Both in the enclosing type definition and in any subtype of the enclosing type, the virtual type definition must be a specialization of its constraint. In a specialization of the enclosing type, a new definition of the virtual type is given by a type definition with the same name and with the keyword **redefined**. A redefined virtual type is still a virtual type. The keyword **finalized** instead of **redefined** indicates that the type is not virtual, and it may thus not be given new definitions in further subtype redefinitions.

A virtual type with an explicit constraint but without an explicit inheritance implicitly inherits from the constraint type.

A redefined or finalized type without an explicit constraint and without an explicit inheritance implicitly inherits from the constraint of the corresponding virtual type.

A subtype of a virtual type is a subtype of the original virtual type and not of a possible redefinition.

6.3.3 Virtual transition/save

This section describes virtual start, input and save as introduced in other sections.

Transitions or saves of a process type, service type or procedure are specified to be virtual transitions or saves by means of the keyword **virtual**. Virtual transitions or saves may be redefined in specializations. This is indicated by transitions or saves respectively with the same (state, signal) and with the keyword **redefined** or **finalized**.

The syntax of virtual transition and save are introduced in 2.4.6 (virtual procedure start), 2.6.2 (virtual process start), 2.6.4 (virtual input), 2.6.5 (virtual save), 2.6.6 (virtual spontaneous transition), 4.10 (virtual priority input), 4.11 (virtual continuous signal), 4.14 (virtual remote procedure input and save).

Virtual transitions or saves must not appear in process (set of instances) definitions or in service (instance) definitions.

A state must not have more than one virtual spontaneous transition.

A redefinition in a specialization marked with **redefined** may in further specializations be defined differently, while a redefinition marked with **finalized** must not be given new definitions in further specializations.

An input or save with <virtuality> must not contain <asterisk>.

Semantics

Redefinition of virtual transitions/saves corresponds closely to redefinition of virtual types (see 6.3.2).

A virtual start transition can be redefined to a new start transition.

A virtual priority input or input transition can be redefined to a new priority input or input transition or to a save.

A virtual save can be redefined to a priority input, an input transition or a save.

A virtual spontaneous transition can be redefined to a new spontaneous transition.

A virtual continuous transition can be redefined to a new continuous transition. The redefinition is indicated by the same (state,[priority]) as the redefined continuous transition. If several virtual continuous transitions exist in a state, then each of these must have a distinct priority. If only one virtual continuous transition exists in a state, the priority may be omitted.

A transition of a virtual remote procedure input transition can be redefined to a new remote procedure input transition or to a remote procedure save.

A virtual remote procedure save can be redefined to a remote procedure input transition or a remote procedure save.

The transformation for virtual input transition applies for virtual remote procedure input transition also.

The transformation of virtual transitions and saves in asterisk state is elaborated in 7.1, step 14.

6.4 Examples

Figure 6.4.1 shows a package “lib”. Figure 6.4.2 shows a block type (enclosed in a package) which uses this package “lib” and another package “slib” which is referenced. Figure 6.4.3 shows a system diagram with a **use** clause.

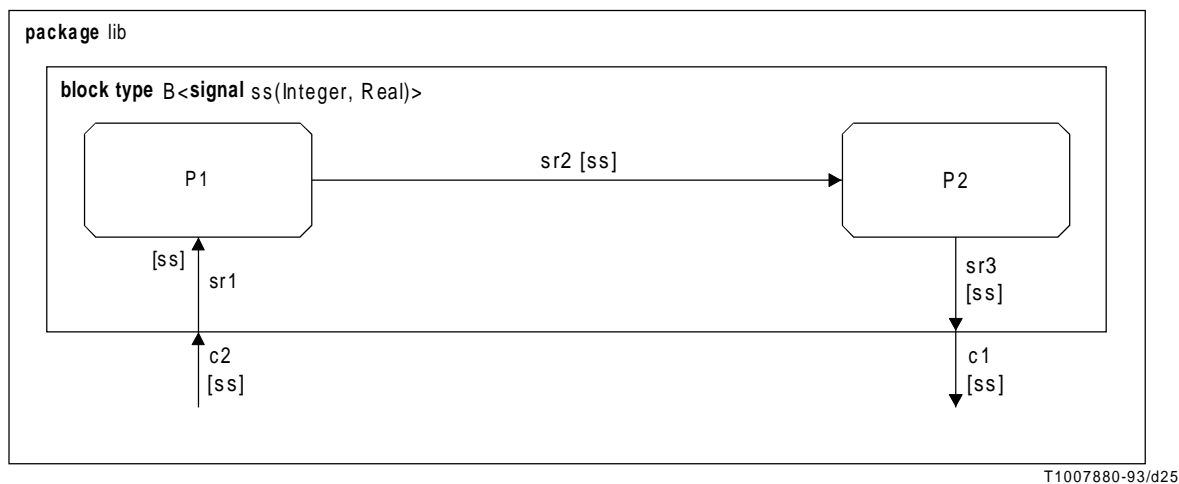


FIGURE 6.4.1/Z.100
Package diagram (SDL/G R)

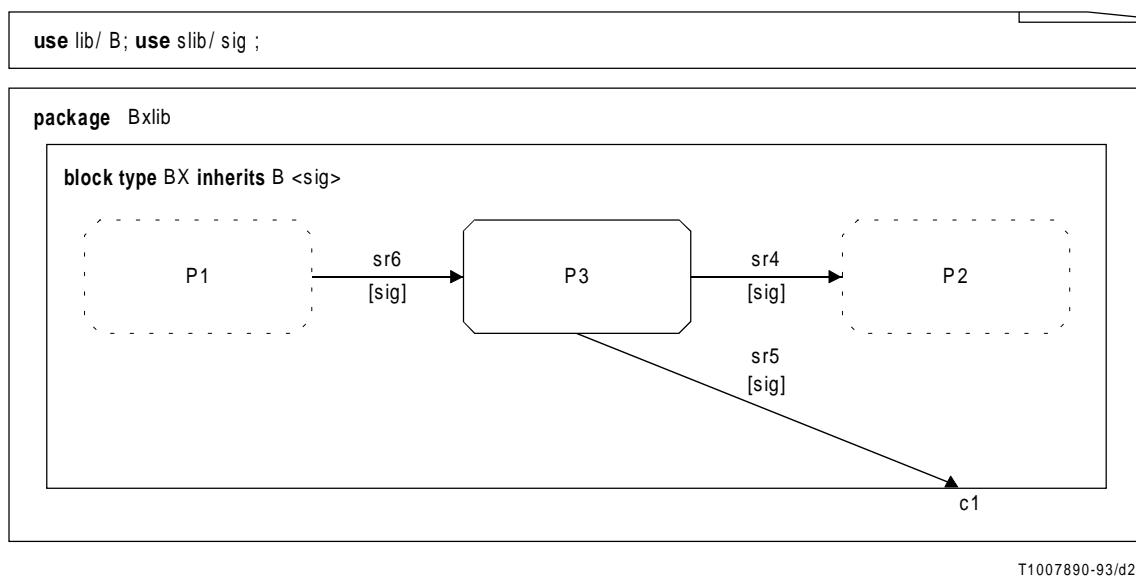
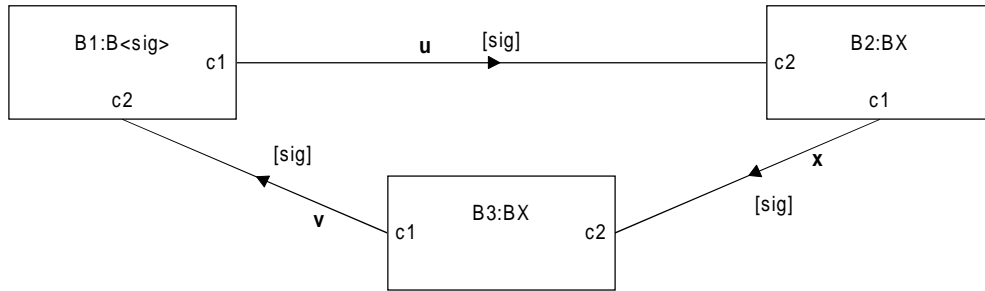


FIGURE 6.4.2/Z.100
Package diagram with use clause (SDL/G R)

```
use lib /B; use BXlib/BX; use slib /sig;
```

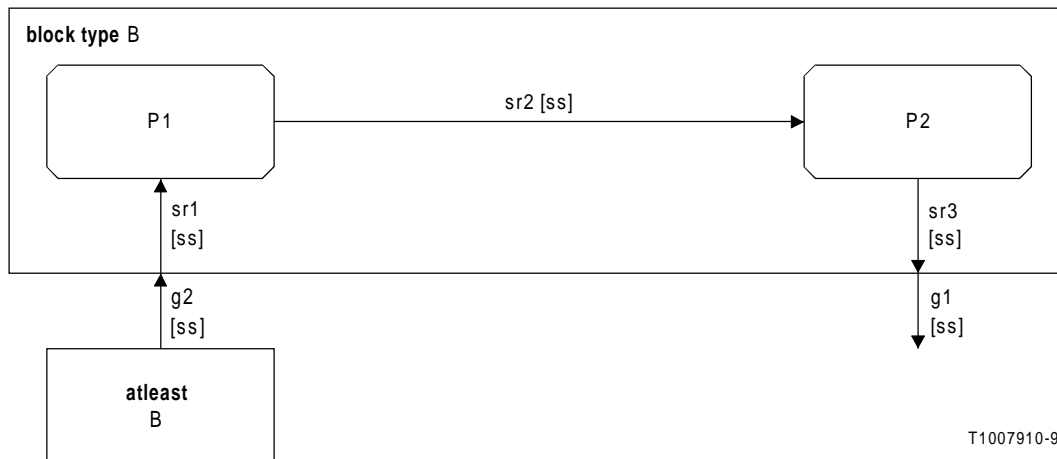
system S



T1007900-93/d27

FIGURE 6.4.3/Z.100

System diagram with use clause (SDL/GR)



T1007910-93/d28

FIGURE 6.4.4/Z.100

Block type with gates g1 and g2 (SDL/GR)

```
block type B;
  gate g1 out with ss;
  gate g2 in from atleast B with ss;
  signalroute sr2 from p1 to p2 with ss;
  signalroute sr1 from env via g2 to p1 with ss;
  signalroute sr3 from p2 to env via g1 with ss;
  process p1 referenced;
  process p2 referenced;
endblock type B;
```

FIGURE 6.4.5/Z.100

Block type with gates g1 and g2 (same as in Figure 6.4.4) (SDL/PR)

Figure 6.4.6 shows a block type BX being a subtype of B with an additional process definition, in SDL/GR, and with references to sets of existing processes (P1 and P3) in order to connect the additional process to them. Figure 6.4.7 shows the same example in SDL/PR.

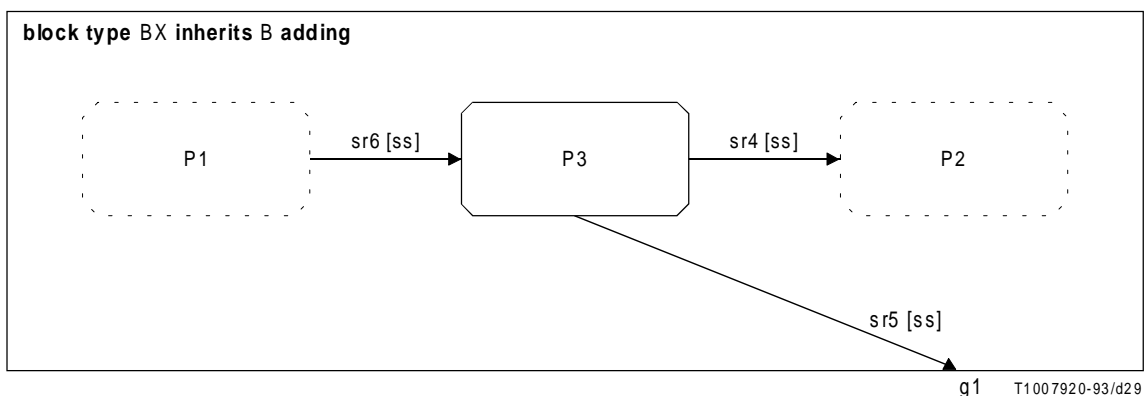


FIGURE 6.4.6/Z.100
Definition of subtype (SDL/GR)

```
block type BX inherits B adding ;
  signalroute sr4 from p3 to p2 with ss;
  signalroute sr5 from p3 to env via g1 with ss;
  signalroute sr6 from p1 to p3 with ss;
process p3 referenced;
endblock type BX;
```

FIGURE 6.4.7/Z.100
Definition of subtype in SDL/PR

Figures 6.4.8 and 6.4.9 show the use of the block types B and BX to specify block instances, and connections to gates.

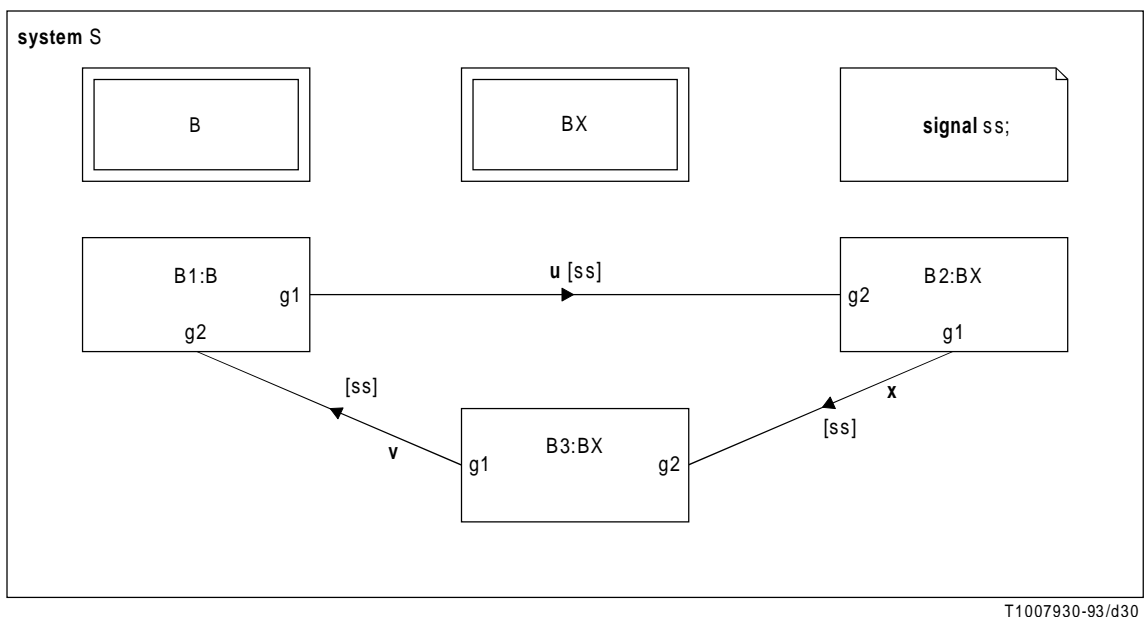


FIGURE 6.4.8/Z.100
Definition of system using block types (SDL/GR)

```

system S;
  signal ss;
  block type B referenced;
  block type BX referenced;

  block B1:B;
  block B2:BX;
  block B3:BX;

  channel u from B1 via g1 to B2 via g2 with ss endchannel;
  channel v from B3 via g1 to B1 via g2 with ss endchannel;
  channel x from B2 via g1 to B3 via g2 with ss endchannel;

endsystem S;

```

FIGURE 6.4.9/Z.100

Definition of system using block types (SDL/PR)

Examples of virtual types are shown in Figure 6.4.10 below:

```

procedure Proc; fpar i,j Integer ... endprocedure;

process type P
  virtual procedure VProc1 atleast Proc
    inherits Proc ... endprocedure;

  virtual procedure VProc2 ... endprocedure;

  ... call VProc1(1,2) ; call VProc2; ...
endprocess type ;

process type P1 inherits P;

  redefined procedure VProc1 atleast VProc1
    inherits << process type P >> VProc1;
  ...
  endprocedure;

  finalized procedure VProc2 ... endprocedure;
  ...
endprocess type ;

process type P2 inherits P1;

  finalized procedure VProc1
    inherits << process type P1 >> VProc1;
  ...
  endprocedure;
  ...
endprocess type ;

```

FIGURE 6.4.10/Z.100

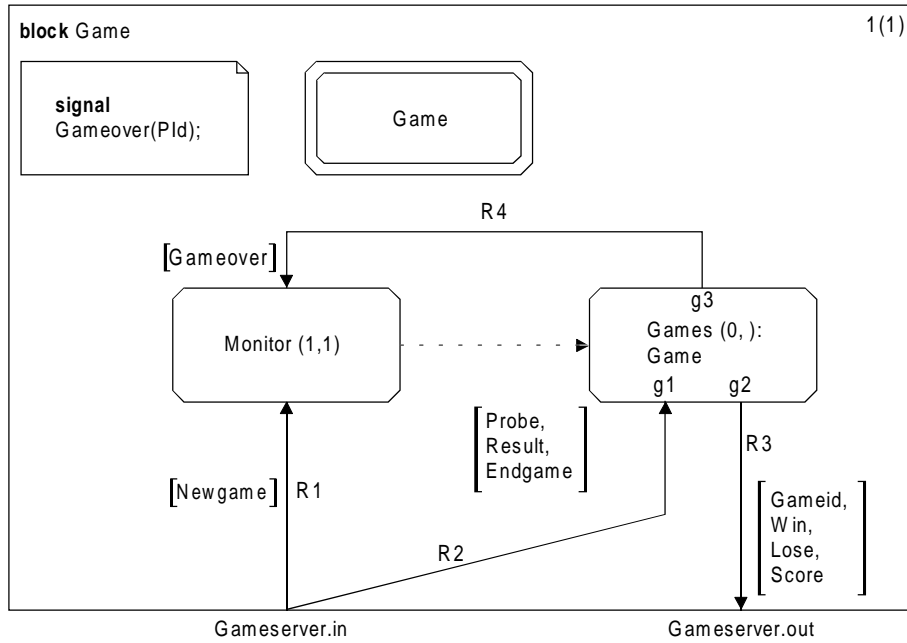
Examples of virtual types (SDL/PR)

In P the procedure VProc1 is virtual, it has to be a specialization of Proc, and the definition of VProc1 inherits from Proc. If a P process is generated, or if a specialized process does not supply a definition of VProc1, then this definition applies. VProc2 is also virtual in P, with an implicit constraint, being itself, i.e. redefinitions of VProc2 in subtypes of P must be subtypes of VProc2.

In P1 the procedure VProc1 is still virtual, constrained by itself, while VProc2 is not virtual. The definitions apply in case a P1 process is generated, also for the calls made in the P definition. The definition of VProc1 is a specialization of the VProc1 defined in process P, therefore it has the qualifier (**process type P**).

In P2 a new definition of VProc1 is given. This is possible since VProc1 is virtual in P1. A redefinition of VProc2 is not allowed here, since it was finalized in P1.

Figures 6.4.11 to 6.4.16 show examples of types corresponding to the example introduced in Figures 2.10.5 to 2.5.10.



T1007940-93/d31

FIGURE 6.4.11/Z.100
Block with process type (SDL/GR)

```

block Game;
  signal Gameover(PId);
  connect Gameserver.in and R1,R2;
  connect Gameserver.out and R3;
  signalroute R1 from env to Monitor with Newgame;
  signalroute R2 from env to Games via g1 with Probe, Result, Endgame;
  signalroute R3 from Games via g2 to env
    with Gameid, Win, Lose, Score;
  signalroute R4 from Games via g3 to Monitor with Gameover;

  process type Game referenced;

  process Monitor (1,1) referenced;

  process Games (0,) : Game referenced;

endblock Game;

```

FIGURE 6.4.12/Z.100
Block with process type (SDL/PR)

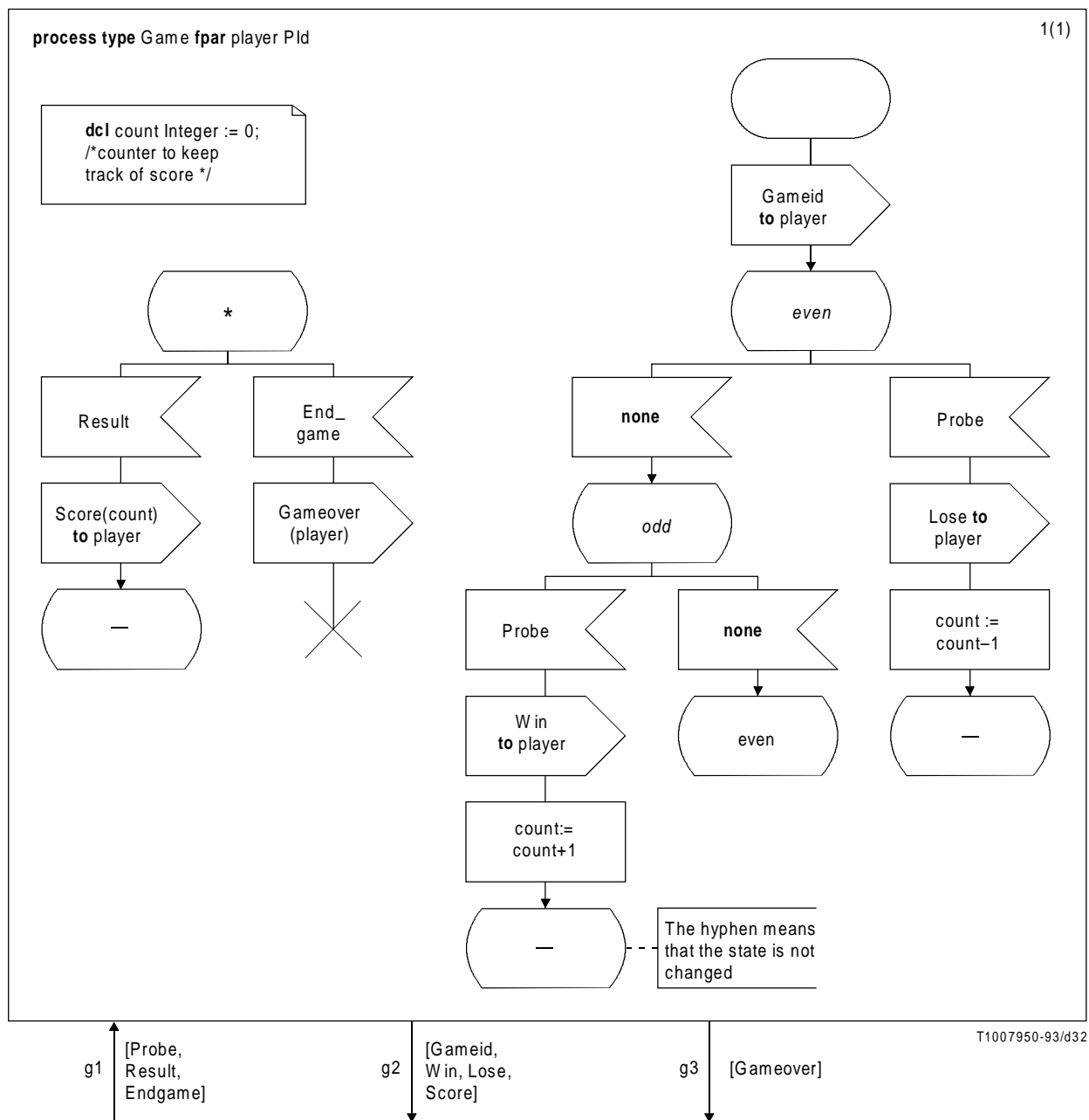


FIGURE 6.4.13/Z.100
Game process as a process type (SDL/GR)

```

process type Game;
  fpar player Pid;

  gate g1 in with Probe, Result, Endgame;
    g2 out with Gameid, Win, Lose, Score;
    g3 out with Gameover;
  dcl count Integer:=0;

  start;
output Gameid to player;
  nextstate Even;
  state Even;
  input none;
    nextstate Odd;
  input Probe;
    output Lose to player;
    task count:= count - 1;
    nextstate -;

  state Odd;
  input none;
    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 Gameover(player);
  stop;

endprocess type Game;

```

FIGURE 6.4.14/Z.100

Game process as a process type (SDL/PR)

Figures 6.4.15 and 6.4.16 show a specialization, SpecialGame of the process type Game, taking into account a new signal Evil from a new process, Devil. In any of the states of Game, the reception of this signal leads to the state Even, allowing the player a greater probability of losing. Note that the Game-part still behaves exactly like a Game process. If the Devil does not send any Evil signals, then the behaviour is the same as in an instance of the supertype, Game.

Note that the extension of the Game process type requires extensions to the block enclosing instances of the specialized process type. Evil is assumed to come in on the same gate as Probe, Result and Endgame.

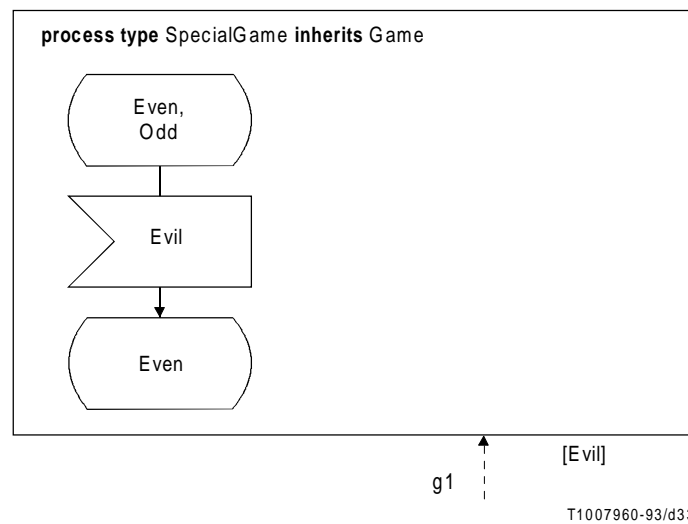


FIGURE 6.4.15/Z.100
Specialization of the Game process (SDL/GR)

```

process type SpecialGame inherits Game;

gate g1 adding in with Evil;

    state Even, Odd;
    input Evil;
    nextstate Even;

endprocess type SpecialGame;

```

FIGURE 6.4.16/Z.100

Specialization of the Game process (SDL/PR)

Figure 6.4.17 shows examples of context parameters. In a), the actual sort must be a specialization of the (at this point visible) sort *t*. In b) it suffices that the first actual parameter is a subtype of *t*, and that the second actual parameter is a sort with an operator accepting two values of the first actual parameter and returning a value of the second parameter. In c) the first actual parameter must be a subtype of *SuperProc* whereas the second actual parameter can be any procedure, which accepts two (**in**) parameters of sort *Integer* and *Boolean*.

```

signal S<newtype t1 atleast t> (t1);      /* case a) */

procedure P<newtype t1 atleast t;          /* case b) */
    newtype t2 operators op: t1,t1 -> t2 endnewtype>;
    dcl x11,x12 t1, x2 t2;
    start;
    task x2:=op(x11,x12);
    return;
endprocedure;

procedure P                                /* case c) */
    < procedure Proc1 atleast SuperProc;
    procedure Proc2 fpar Integer, Boolean >;
    start;
    call Proc1;
    call Proc2(7, True);
    return;
endprocedure;

```

FIGURE 6.4.17/Z.100

Examples of context parameters (SDL/PR)

7 Transformation of SDL Shorthands

This section details the transformation of the SDL constructs, whose dynamic semantics are given after a transformation to the subset of SDL for which *Abstract Grammar* exists. These shorthand notations are

- a) constructs from clauses 2, 3 and 5 for which a *Model* section exists, and
- b) constructs defined in clauses 4 and 6.

The properties of a shorthand notation are derived from the way it is modelled 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 detailed 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.

The transformations are described as a number of enumerated steps. One step may describe the transformation of several concepts and thus consist of a number of sub-steps, either because these concepts must be transformed as a group or because the transformation order between these concepts is not significant. The latter case, is indicated by a dash (–) rather than by enumeration.

7.1 Transformation of additional concepts

1. Lexical transformations
 - 1) `<macro definition>s` and `<macro call>s` (4.2) are identified lexically and `<macro call>s` are expanded.
 - 2) `<macro diagram>s` are replaced by occurrences of `<macro call area>`.
 - 3) `<underline>` followed by separator characters are removed from names and separators in names are replaced by an `<underline>` (2.2.1). After this transformation the `<sdl specification>` is considered to be syntactically well-formed.
 - 4) `<macro definition>s` are removed (also in `<package definition>s`).
2. Definition references are replaced by `<referenced definition>s` (2.4.1.3).
3. The graphs are normalized, i.e.
 - 1) Non-terminating decisions and non-terminating transition options are transformed into terminating decisions and terminating transition options respectively.
 - 2) The actions and/or terminator statement following the decisions and transition options are moved to appear as `<free action>s`. Those generated `<free action>s` which have no label attached are given anonymous labels.
 - 3) Action lists (including the terminator statement which follows) where the first action (if any, otherwise the following terminator statement) has a label attached, are replaced by a join to the label and the action list appears as a `<free action>`.
4. The package Predefined is included in the `<package list>`.
5. Transformation of generic system (4.3) and external data (5.4.6):
 - 1) Identifiers in `<simple expression>s` contained in the `<sdl specification>` are bound to definitions. During this binding, only `<data definition>s` defined in the predefined package Predefined and `<external synonym definition>s` are considered (i.e. all other `<data definition>s` are ignored).

- 2) <external synonym>s are replaced by <synonym definition>s and informal text in transition options is replaced by <range condition>. How this is done is not defined by SDL.
- 3) <simple expression>s are evaluated and <select definition>s, <option area>s, <transition option>s and <transition option area>s are removed. <transition>s which are not selected in a <transition option> or <transition option area> are given anonymous labels and appear then as <free action>s or <in-connector area> respectively.

The values in <number of process instances> are represented by fully qualified Integer literal identifiers.

- 4) <external properties> are replaced by axioms and informal text. How this is done is not part of SDL.

6. Package inclusion (2.4.1.2)

<package>s which are mentioned in a <package reference clause>, but which are not part of the <package list> are included in the <package list>. How this is done is not defined by SDL.

If the <sdl specification> does not contain a <system definition> no further steps are applied, though the properties of packages assures that further steps in practice can be applied.

7. Transformation of:

- Multiple assignments in <task body> (2.7.1)
- Non-deterministic decision (2.7.5)
- <infix operator>s and their operands transformed to the prefix form (5.4.1.1)
- <procedure result> (2.4.6)
- Structure sort (5.3.1.10)
- State list (2.6.3)
- Stimulus list (2.6.4)
- Field primary (5.3.2.5)
- Structure primary (5.3.2.6)
- Boolean axioms (5.3.1.5)
- <syntype definition>s with **newtype** (5.3.1.9)
- Multiple signals in <output body> (2.7.4)
- Multiple timers in <set> and <reset> (2.8)

8. Full qualifiers are inserted

According to the visibility rules and the rules for resolution by context (2.2.2), qualifiers are extended to denote the full path.

NOTE – As some additional concepts (e.g. generator transformations and specialization) have impact on the set of definitions attached to a scope unit, the qualifiers are derived after removal of such concepts for the particular scope unit.

For each scope unit:

- Context parameters (6.2)
- Specialization (6.3)
- Signal list (2.5.5)
- Ordering (5.3.1.8)
- Generators (5.3.1.12)
- Indexed primary (5.3.3.4)
- Field variable (5.4.3.2)
- Indexed variable (5.4.3.1)
- Input of fields (2.6.4)
- Open block substructure diagram (3.2.2)
- Default duration value for timer set (2.8)
- Initialization of variables of sorts with default initialization (5.4.3.3)

are transformed, whereupon qualifiers are inserted. Afterwards, the transformations are applied for scope units enclosed in the scope unit.

The step is defined separately in 7.2 below.

9. Implicit specialization of global procedures (2.7.3)

If an occurrence of a <procedure identifier> refers to a procedure which is not enclosed by a process or service, it causes a copy local to the process to be made and the <procedure identifier> to be changed to denote the copy. This step is repeated until there are no more references inside processes or services to non-local procedures.

<operator definition>s are transformed into procedures having anonymous names and having the result as **in/out** parameter. The resulting <procedure definition> is moved to the enclosing scope unit, the operators signatures for each operator are removed and the operator applications are transformed into <value returning procedure call>s.

10. Transformation of:

- <textual typebased system definition>s, <graphical typebased system definition>s, <textual typebased block definition>s, <graphical typebased block definition>s, <textual typebased process definition>s, <graphical typebased process definition>s, <textual typebased service definition>s and <graphical typebased service definition>s are replaced by <system definition>s, <system diagram>, <block definition>, <block diagram>, <process definition>, <process diagram>, <service definition> and <service diagram> respectively, by copying the contents of the type denoted by the <base type>. In this copy, <formal context parameter>s are replaced by the <identifier>s in <actual context parameters> in the same manner as described in sub-step 2.2 of step 8 (see below).
- The keyword **this** in procedure call, is replaced by the procedure identifier.
- Occurrences of type names inside qualifiers are replaced by the respective instance names and occurrences of **this** where it denotes a <process identifier> are replaced by the identifier denoting the enclosing <process definition> or <process diagram>.
- <channel connection>s, <channel to route connection>s, <signal route to route connection>s, the full signalset and omitted signal routes (if any) are derived from the <gate>s occurring in channels and signal routes connected to the instance.

These transformations apply both for <process definition>s and <service definition>s.

- <gate>s in <output body>s are replaced by those channels or signal routes which are connected to the instance, which conveys the signal and which mentions the gate.
- The <gate>s in channels and signal routes are removed.
- Block sets are removed and the **via** is modified:

Each channel definition having a block set as endpoint is replaced by a number of channels, one for each block in the set. Each new channel is connected to a copy of the block definition defined by the block set. The copy has a distinct new name.

For **via** and connections contained inside a block copy, an occurrence of the original channel identifier is replaced by the channel identifier connecting that particular block member.

For **via** and connections contained inside a block connected to a block set, all occurrences of the original channel identifier is replaced by the list of channel identifiers resulting from expanding the block set.

If the two endpoints of the original channel denotes the same <block identifier>, the resulting number of channel equals the square of the number of block instances minus the number of block instances.

11. Removal of:
 - 1) Types having <formal context parameter>s, system types, block types, process types, service types, occurrences of <specialization> and <inheritance rule>.
 - 2) Procedures defined outside processes or services and <virtuality> for remaining procedures.
12. The definitions inside packages (2.4.1.2) are moved to the system level.

 Each name defined inside a package will be renamed to an anonymous name and qualifiers are changed to denote the system level.

 The package list is removed.
13. Transformation of channel substructure (3.2.3).
14. Removal of asterisk state, asterisk input and asterisk save:
 - A body originating from a process definition, service definition or procedure definition without specialization, has its asterisk states expanded according to the model defined in 4.4 followed by expansion of asterisk input according to the model defined in 4.6, followed by the model for asterisk save according to the model defined in 4.7.
 - A body which is formed by combining two or more type bodies (through specialization) retains information of which type body its states originate from. That is, it can be regarded as consisting of a list of type bodies, where the first type body originates from the type having no specialization, and type body N (N>1) in the list originates from the process type body being a specialization of type body N-1.
 - The combined body is formed by applying the following steps:
 - 1) Include all states from all bodies, but exclude associated transitions and saves, which are redefined or finalized.
 - 2) Expand asterisk state like for a body originating from a definition without specialization.
 - 3) Expand asterisk input and asterisk save like for a body originating from a definition without specialization.
 - 4) Replace virtual and redefined transitions and saves with the redefined and finalized transitions and saves for the types one by one in ascending order of N. If a state is an asterisk state, the replacement applies for all states in the combined body.
 - Multiple appearance of state is merged (4.5).
15. Implicit transitions are inserted (4.8).
16. **via all** is changed into an output list (2.7.4).
17. <free action>s (2.6.7) are removed:
 - 1) Any <join> or <out-connector area> is replaced by the <free action> or <in-connector area> respectively containing that connector name, unless the <join> or <out-connector area> is contained in the same <free action> or <in-connector area> as the corresponding connector name.
 - 2) The <free action>s and <in-connector area>s are then discarded. The resulting graphs are no longer acyclic apart from loops inside a transition (no transitions are shared by inputs).
18. Dash nextstate is replaced by a state name (4.9).
19. Trailing commas in <stimulus> (2.6.4), <create body> (2.7.2), <procedure body> (2.7.3) and <output body> are inserted.

20. Synonym identifiers are replaced by the expression they denote (5.3.2.3).
21. Priority inputs are transformed (4.10).
22. Continuous signal is transformed (4.11).
23. Enabling condition is transformed (4.12).
24. Implicit tasks for expressions containing <now expression> (5.4.4.2), <import expression> (4.13), <view expression> (5.4.4.4), <timer active expression> (5.4.4.5) and implicit procedure calls for expressions containing <value returning procedure call>s (5.4.5) are inserted.
25. Imported and exported values (4.13) and remote procedures (4.14) are transformed.

7.2 Insertion of full qualifiers

This section details the contents of step 8 (see 7.1 above), full qualifiers:

Insertion of full qualifiers and transformation of the concepts transformed in this section are done package by package (in the order given by <package list>) and ending with the <system definition>. Insertion of full qualifiers is done by applying the visibility rules and resolution by context rules as defined in 2.2.2.

Steps 1 to 9 below are repeated until all scope units have been transformed.

1. Select a scope unit (initially the leftmost <package> in <package list> or the <system definition>), which satisfies the following conditions:
 - 1) The surrounding scope unit (if any) has been transformed.
 - 2) Omitted <actual context parameter>s are inserted (6.1.2).
 - 3) If it contains <specialization> then any scope unit contained in that type given by <base type> has been transformed.
 - 4) It has not been transformed already.
2. If the scope unit contains <specialization>, make a copy of the <base type> and bind any <actual context parameter>s contained in the <specialization>. The copy of the type denoted by <base type> is made such that:
 - 1) Virtual types which also have a definition in the scope unit to be transformed, are given unique anonymous names, but the scope units keep knowledge of their original full qualifier such that identifiers denoting the original virtual types (i.e. those identifiers qualified with the <base type> inside the scope unit and inside specializations of the scope unit) can be replaced by the new identity when qualifiers are inserted (step 7 below).
 - 2) Any applied occurrences (except in generators) of <formal context parameter>s defined for that type are replaced by corresponding <actual context parameter>s. Subsignals, literals and operators of <signal context parameter>s and of <sort context parameter>s respectively have their qualifier changed to denote the identifier of the <actual context parameter>. <formal context parameter>s corresponding to omitted <actual context parameters> are not replaced but instead become (parts of the) <formal context parameter>s of the scope unit.
 - 3) Virtual transitions are not transformed at this stage, as there is an interdependency with asterisk input, asterisk save and asterisk state. Instead knowledge in the combined body is inserted about which type body any state originates from. Consequently, the combined body cannot be regarded valid until asterisk input, asterisk save and asterisk state have been transformed (see step 14, in 7.1).
 - 4) Occurrences of the type name in qualifiers contained in the type are replaced with the name of the scope unit (except in generators).
3. Transformation of <generator transformation>

- 1) <generator transformation>s inside <generator definition>s occurring in the scope unit are transformed.
- 2) <generator transformation>s inside <partial type definition>s occurring in the scope unit are transformed.
4. <ordering> is transformed (5.3.1.8).
5. <partial type definition>s are transformed:
 - 1) If a <partial type definition> contains <type expression>, the literals and operators are copied according to the inheritance list. If the <base type> for a <partial type definition> has formal <sort context parameter>s, any argument sort or result sort mentioning a <sort context parameter> is changed accordingly.
 - 2) The equality properties are derived (5.3.1.4).
 - 3) Qualifiers of identifiers in <axioms>, in the <range condition>s and in the default expression are inserted and explicit quantification is inserted. Resolution by context is done by discarding any <partial type definition>s having <formal context parameter>s (except for the enclosing one, if any).
 - 4) For any <partial type definition> containing a <type expression> with <actual context parameter>s, the <axioms>, the <default initialization> and the <range condition>s contained in the <base type> are copied to the <partial type definition>, where <formal context parameter>s have been replaced by the corresponding <actual context parameter>s.
 - 5) For any <partial type definition> containing a <type expression> with no <actual context parameter>s, the implied axioms are inserted (5.3.1.11).
6. <signal definition>s with <specialization> are removed:

A signal definition which has <specialization> and where the <base type> denotes a <signal definition> without <specialization> is selected. The model for <specialization> for this <signal definition> is applied in a similar manner as step 2 above. This step is repeated until there are no more <signal definition>s having <specialization> in the scope unit.
7. Qualifiers in those identifiers directly contained in the scope unit and in those contained in contained signal definitions are inserted.

Resolution by context is done by discarding any <partial type definition>s having <formal context parameter>s.

References directly contained in the scope unit to a virtual type in a supertype of an enclosing scope unit are transformed into references to the anonymous copy (see step 2.1 above).
8. <stimulus> containing <indexed variable>s and <field variable>s are transformed (2.6.4).
9. Transformation of:
 - Signal list identifiers to a list of signal identifiers (2.5.5)
 - Indexed primary (5.3.2.4)
 - Field variable (5.4.3.2)
 - Indexed variable (5.4.3.1)
 - <return> with <expression> (2.4.6)

Annex A

Index

to clauses 2 to 7 of Recommendation Z.100 (normative parts)

(This annex forms an integral part of this Recommendation)

The entries are:

- a) <keyword>s from the concrete grammar.
- b) Non-terminals from the concrete grammar. These are enclosed in angle brackets (i.e. < and >). The definitions are the bolded index entries.
- c) Non-terminals from the abstract grammar. The definitions are the bolded index entries.
- d) Glossary items, these are the words marked with (G). Only glossary entries which refer to clauses 2 to 7 are included.

If both a defining and an applying occurrence of a non-terminal exist on the same page, the bolded entry for the defining occurrence takes priority. Non-terminals from the concrete grammar who have no bolded entry are either

- 1) not part of syntax rules; or
- 2) contain underlined parts (underlining is not shown in the index).

Non-terminals containing underlined parts should be looked up ignoring the underlined parts for a defining occurrence.

<action statement>; 57; 58; 60; 68; 69; 101
<action>; 55; 58
<active alternative expression>; 156
<active consequence expression>; 156
<active expression list>; 156; **157**
<active expression>; **154**; 156; 157
<active extended primary>; 154
<active primary>; 149; 150; **154**
<actual context parameter>; 19; 172; 177; **178**; 198; 199
<actual context parameters>; 137; 138; 171; **177**; 178; 183; 196; 198
<actual parameters>; 63; 64; 65; 66; 67; 112; 164
<additional heading>; 21
<alphanumeric>; 14; 15; 128
<alternative expression>; 156
<alternative ground expression>; 153
<alternative question>; 100; 101
<alternative>; 130
<answer part>; 68; 69; 100; 101; 106; 108
<answer>; 68; 69; 70; 101
<any area>; 92; 94; 95
<anyvalue expression>; 69; 160; **163**
<apostrophe>; 15; 22; 128
<argument list>; 119; 120; 133; 136
<argument sort>; 19; 119; 120; 121; 133; 147; 150
<assignment statement>; 55; 62; **157**; 158; 159
<asterisk input list>; 54; **103**
<asterisk save list>; 55; 103
<asterisk state list>; 52; **102**
<asterisk>; 102; 103; 185
<axioms>; 117; 118; **122**; 141; 145; 199
<base type literal rename signature>; 137; 138
<base type operator name>; 137
<base type>; 49; 138; **171**; 174; 176; 178; 179; 182; 183; 196; 198; 199
<basic input area>; **54**; 55

<basic input part>; **54**; 55
 <basic save area>; **55**; 56
 <basic save part>; **55**; 56
 <block area>; **29**; 43; 92; 99
 <block definition>; 18; 27; 28; **30**; 31; 32; 34; 43; 46; 47; 84; 87; 88; 96; 98; 118; 173; 177; 196
 <block diagram>; 18; 27; 29; **31**; 32; 85; 196
 <block heading>; **31**
 <block identifier>; 30; 31; 43; 87; 88; 173; 196
 <block interaction area>; **29**; 85; 87; 92; 96; 166
 <block name>; 28; 29; 30; 31; 168; 173
 <block substructure area>; 31; **85**; 92; 168
 <block substructure definition>; 18; 27; 30; **84**; 85; 86; 88; 118; 167; 176; 177; 183
 <block substructure diagram>; 18; 27; 43; **85**; 86
 <block substructure heading>; **85**
 <block substructure identifier>; 84; 85
 <block substructure name>; 84; 85
 <block substructure symbol>; **85**
 <block substructure text area>; **85**; 92; 99
 <block symbol>; **29**; 85; 88; 173; 176
 <block text area>; **31**; 92; 99; 168
 <block type definition>; 18; 24; 27; 28; 31; 32; 43; 46; 47; 84; 85; 98; 118; **167**; 168
 <block type diagram>; 18; 27; 46; 85; 99; 167; **168**
 <block type expression>; 173
 <block type heading>; **168**
 <block type identifier>; 167; 168
 <block type name>; 167; 168
 <block type reference>; 92; 99; 167; **168**
 <block type symbol>; 167; **168**
 <body>; 19; **57**; 60; 102; 103; 183
 <Boolean active expression>; 156
 <Boolean axiom>; 122; **129**
 <Boolean expression>; 105; 106; 107; 108; 156; 163
 <Boolean ground expression>; 153
 <Boolean simple expression>; 98; 99
 <Boolean term>; 129; 130
 <channel connection>; **84**; 85; 88; 90; 98; 177; 196
 <channel definition area>; 29; **43**; 92; 99
 <channel definition>; 28; **43**; 84; 87; 88; 98
 <channel endpoint connection>; **87**; 88; 90; 98
 <channel endpoint>; **43**
 <channel identifier>; 48; 66; 88
 <channel identifiers>; 31; 43; 46; **48**; 84; 85; 177
 <channel name>; 43; 87
 <channel path>; **43**; 177
 <channel substructure area>; **87**; 92
 <channel substructure association area>; 43; **87**; 93
 <channel substructure definition>; 18; 27; 43; **87**; 88; 118
 <channel substructure diagram>; 18; 27; 43; **87**; 88
 <channel substructure heading>; **87**
 <channel substructure identifier>; 87
 <channel substructure name>; 87
 <channel substructure symbol>; 87; **88**
 <channel substructure text area>; **87**; 93; 99
 <channel symbol 1>; **43**
 <channel symbol 2>; **43**
 <channel symbol 3>; 43; **44**
 <channel symbol 4>; 43; **44**
 <channel symbol 5>; 43; **44**
 <channel symbol>; 20; **43**; 44; 85; 87; 92; 95
 <channel to route connection>; 30; 47; **48**; 98; 177; 196
 <character string literal identifier>; 126; **127**
 <character string literal>; 126; **127**; 128; 137; 143; 144; 146
 <character string>; 14; **15**; 16; 17; 19; 20; 21; 22; 68; 95; 127; 128

<closed range>; **135**
 <comment area>; **21**; 93
 <comment symbol>; 20; **21**; **22**
 <comment>; **21**
 <composite special>; 14; **15**
 <composite term list>; **122**; 123; 125
 <composite term>; **122**; 125
 <condition>; **130**
 <conditional composite term>; 125; **130**
 <conditional equation>; 122; 123; **124**
 <conditional expression>; 154; **156**
 <conditional ground expression>; 149; **153**; 156
 <conditional ground term>; 126; **130**
 <conditional term>; **130**
 <connector name>; 19; 57; 60; 183
 <consequence expression>; **156**
 <consequence ground expression>; **153**
 <consequence>; **130**
 <constant>; 70; **135**
 <context parameters end>; 177; **178**
 <context parameters start>; 177; **178**
 <continuous signal area>; 93; **105**; 106; 163
 <continuous signal association area>; 53; 93; **105**
 <continuous signal>; 52; **105**; 106; 108; 163
 <create body>; **63**; 197
 <create line area>; **31**; 93; 99
 <create line symbol>; 20; **31**; 32; 92; 95
 <create request area>; 59; **63**; 93
 <create request symbol>; **63**
 <create request>; 58; **63**
 <dash nextstate>; 59; **104**
 <dashed association symbol>; 20; **21**; **22**; 87; 92; 95
 <dashed block symbol>; **173**
 <dashed process symbol>; **174**
 <dashed service symbol>; 174; **175**
 <data definition>; 17; 24; 28; 29; 30; 33; 35; 37; 38; 40; 98; 147; 148; **153**; 194
 <decimal digit>; **14**
 <decision area>; 59; **69**; 93
 <decision body>; **68**; 69
 <decision symbol>; **69**
 <decision>; 58; **68**; 69; 106; 108
 <default initialization>; 51; 117; 118; 133; 138; 140; **159**; 199
 <definition selection list>; **24**; 25; 26
 <definition selection>; **24**; 25; 26
 <definition>; 26; **27**
 <destination>; **66**; 110; 111; 112; 114
 <diagram identifier>; 21
 <diagram in package>; **25**
 <diagram kind>; 21
 <diagram name>; 21
 <diagram>; 22; 26; **27**
 <dummy inlet symbol>; 94; **95**; 96
 <dummy outlet symbol>; **92**; 94; 95; 96
 <Duration ground expression>; 70; 71
 <else part>; **68**; 69; 100; 101
 <enabling condition area>; 54; 56; 93; **107**; 113; 163
 <enabling condition symbol>; 105; **107**
 <enabling condition>; 54; 56; **107**; 108; 112; 163
 <end>; **21**; 24; 28; 30; 31; 33; 35; 37; 39; 43; 45; 48; 49; 50; 51; 52; 54; 55; 56; 57; 58; 68; 70; 84; 87; 91; 95; 98; 100;
 104; 105; 107; 110; 112; 113; 119; 122; 136; 137; 141; 144; 147; 153; 159; 164; 166; 167; 168; 169; 170; 172; 173;
 174; 175; 177; 179
 <endpoint constraint>; **176**
 <entity in block>; **30**; 167

<entity in package>; **24**; 25
 <entity in procedure>; 39; **40**
 <entity in process>; **33**; 168
 <entity in service>; **37**; 170
 <entity in system>; **28**; 84; 87; 166
 <entity kind>; **24**; 25; 26
 <equation>; **122**; 145
 <error term>; 122; **131**
 <exclamation>; 19; **126**; 131; 140
 <existing gate symbol>; **176**; 183
 <existing typebased block definition>; 29; 93; **173**
 <existing typebased process definition>; 31; **174**
 <existing typebased service definition>; 35; **174**; 175
 <export area>; 59; 93; **110**
 <export>; 58; **110**; 111; 112
 <exported as>; **50**
 <exported variable definition>; 110
 <expression list>; 70; 151; 152; **154**; 157; 158; 162; 163
 <expression>; 42; 61; 63; 64; 65; 67; 101; 127; 148; **149**; 150; 154; 156; 157; 158; 159; 164; 199
 <extended composite term>; 122; **125**; 126
 <extended ground term>; 122; **126**
 <extended literal identifier>; 122; **126**
 <extended literal name>; 119; 120; **126**
 <extended operator identifier>; 125; **126**
 <extended operator name>; 119; 120; 123; **126**
 <extended primary>; **149**; 150; 154
 <extended properties>; 117; 118; **125**; 133
 <extended sort>; 119; **120**; 122; 140; 141; 145; 147
 <external data description>; **164**
 <external formalism name>; **164**
 <external properties>; 117; 118; **164**; 195
 <external signal route identifiers>; 46; **48**; 177
 <external synonym definition item>; **97**
 <external synonym definition>; **97**; 142; 194
 <external synonym identifier>; 97
 <external synonym name>; 97
 <external synonym>; **97**; 150; 195
 <field extract operator name>; 152
 <field list>; **136**
 <field modify operator name>; 159
 <field name>; 136; 151; 152; 158; 159
 <field primary>; 149; 150; **151**
 <field selection>; **151**; 158
 <field sort>; **136**
 <field variable>; 55; 157; **158**; 159; 199
 <fields>; **136**
 <first constant>; 135
 <first field name>; 152
 <flow line symbol>; 20; 57; **60**; 69; 92; 95; 101
 <formal context parameter>; 19; 49; 118; 166; 167; 169; 170; 172; 177; **178**; 179; 196; 197; 198; 199
 <formal context parameters>; 39; 41; 49; 117; 118; 166; 167; 168; 169; 170; **177**; 178
 <formal name>; **91**; 96
 <formal parameters signature>; **179**
 <formal parameters>; 19; **33**; 35; 147; 148; 168; 169
 <formal variable parameters>; 19; **39**
 <frame symbol>; 20; 21; 25; **29**; 31; 34; 38; 40; 43; 46; 85; 87; 91; 92; 96; 147; 166; 168; 169; 170; 172; 176
 <free action>; 33; 40; **57**; 147; 194; 195; 197
 <full stop>; **14**; 15; 128
 <gate constraint>; **175**; 176
 <gate definition>; 19; 167; 168; 170; **175**
 <gate identifier>; 66
 <gate name>; 19; 175; 176
 <gate symbol 1>; **176**
 <gate symbol 2>; **176**
 <gate symbol>; **176**

<gate>; 43; 45; 46; 168; 169; 170; 173; 174; **175**; 176; 177; 196
 <generator actual list>; **141**
 <generator actual>; **141**
 <generator definition>; **139**; 140; 153; 199
 <generator formal name>; 19; 126; **139**; 140; 141
 <generator identifier>; 141
 <generator name>; 139; 140
 <generator parameter list>; **139**
 <generator parameter>; **139**; 140; 141
 <generator sort>; 120; **140**
 <generator text>; 126; **139**; 140; 141
 <generator transformation>; **141**; 142; 198; 199
 <generator transformations>; 17; 125; 139; **141**
 <graphical answer part>; **69**
 <graphical answer>; **69**; 101
 <graphical block reference>; **29**; 93
 <graphical block substructure reference>; **85**
 <graphical channel substructure reference>; **87**
 <graphical else part>; **69**
 <graphical gate constraint>; 43; 46; 168; 169; 170; **176**
 <graphical procedure reference>; 38; **40**; 93; 167; 168; 169; 170
 <graphical process reference>; **31**; 34; 93
 <graphical service reference>; **35**
 <graphical typebased block definition>; 29; 93; **173**; 196
 <graphical typebased process definition>; 31; **174**; 196
 <graphical typebased service definition>; 35; **174**; 175; 196
 <graphical typebased system definition>; 29; **172**; 196
 <ground expression list>; 149; **150**
 <ground expression>; 50; 51; 97; 101; 135; 142; 147; **149**; 150; 153; 157; 159
 <ground primary>; **149**; 150
 <ground term>; **122**; 126; 141; 145
 <heading area>; 20; **21**
 <heading>; **21**
 <hyphen>; **104**
 <identifier>; **17**; 18; 19; 26; 27; 123; 147; 150; 171; 175; 176; 178; 184; 196
 <imperative operator>; 147; 154; **160**
 <implicit text symbol>; 21
 <import expression>; **110**; 111; 160; 161; 198
 <import identifier>; 111
 <imported procedure specification>; 33; 35; 37; 38; 98; **112**; 113
 <imported variable specification>; 33; 35; 37; 38; 98; **110**; 111
 <in-connector area>; 35; 41; **57**; 60; 93; 147; 169; 195; 197
 <in-connector symbol>; **57**; 60
 <indexed primary>; 149; 150; **151**
 <indexed variable>; 55; 157; **158**; 199
 <infix operator>; 125; 126; **127**; 195
 <informal text>; 16; **20**; 62; 68; 97; 100; 122
 <inheritance list>; 126; 129; **137**; 138; 182
 <inheritance rule>; 125; 129; **137**; 178; 197
 <inherited operator name>; **137**; 138
 <inherited operator>; **137**
 <initial number>; **33**; 34
 <inlet symbol>; 94; **95**; 96
 <input area>; 53; **54**; 93
 <input association area>; **53**; 93
 <input list>; **54**; 103; 105; 107
 <input part>; 52; **54**; 55; 103; 104; 106; 107; 112; 114
 <input symbol>; 20; **54**; 56; 113
 <Integer literal name>; 105; 106
 <interface>; 24; **25**; 26
 <internal input symbol>; 20; 54; **71**

<internal output symbol>; 20; 66; **72**
 <internal properties>; **117**
 <internal synonym definition>; **142**
 <join>; 57; 58; **60**; 69; 101; 197
 <kernel heading>; **21**
 <keyword>; 14; **15**; 17
 <label>; **57**; 58
 <left curly bracket>; **14**
 <left square bracket>; **14**
 <letter>; **14**; 16
 <lexical unit>; **14**; 16; 91; 95
 <literal axioms>; 144; **145**
 <literal equation>; 129; **144**; 145; 146
 <literal identifier>; **122**; 147; 149
 <literal list>; **119**; 132
 <literal mapping>; 117; 118; 141; **144**; 145; 146
 <literal name>; 21
 <literal operator identifier>; 120; 122; 123; 145; 146
 <literal operator name>; 119; 120; 137; 146
 <literal quantification>; 144; **145**; 146
 <literal rename list>; **137**; 138
 <literal rename pair>; **137**
 <literal rename signature>; **137**; 138
 <literal renaming>; **137**; 182
 <literal signature>; 19; **119**; 120; 132; 141; 146
 <macro actual parameter>; 91; 94; **95**; 96
 <macro body area>; 91; **92**
 <macro body port1>; **92**; 96
 <macro body port2>; **92**; 96
 <macro body>; 16; **91**; 94
 <macro call area>; 93; **95**; 96; 194
 <macro call body>; **95**
 <macro call port1>; **95**; 96
 <macro call port2>; **95**; 96
 <macro call symbol>; **95**; 96
 <macro call>; **95**; 96; 194
 <macro definition>; 24; 27; 28; 29; 30; 33; 35; 37; 38; 39; 40; **91**; 94; 98; 147; 194
 <macro diagram>; 25; 27; 29; 31; 34; 38; 40; 85; 87; **91**; 94; 95; 96; 99; 147; 166; 168; 169; 170; 194
 <macro formal name>; 19
 <macro formal parameter>; **91**; 94; 95; 96
 <macro formal parameters>; **91**; 92
 <macro heading>; 91; **92**
 <macro inlet symbol>; **92**; 96
 <macro label>; **92**; 94; 95; 96
 <macro name>; 19; 91; 92; 94; 95; 96
 <macro outlet symbol>; **92**; 96
 <macro parameter>; **91**
 <maximum number>; **33**; 34
 <merge area>; 59; **60**; 93
 <merge symbol>; **60**
 <monadic operator>; 125; 126; **127**
 <name class literal>; 126; 141; **143**
 <name>; 16; **17**; 18; 19; 24; 26; 27; 85; 91; 92; 94; 111; 120; 123
 <national>; **14**; 16
 <Natural literal name>; 21; 143; 144
 <Natural simple expression>; 33; 173
 <nextstate area>; 20; 53; **59**; 93
 <nextstate body>; **59**
 <nextstate>; 58; **59**; 104; 105; 106; 107; 108
 <noequality>; 119; 120; **129**
 <note>; 14; **15**; 16; 21
 <now expression>; **160**; 161; 198
 <number of block instances>; 44; **173**

<number of pages>; **21**
 <number of process instances>; 31; **33**; 34; 35; 173; 195
 <open block substructure diagram>; **85**; 86
 <open range>; **135**
 <operand0>; **149**
 <operand1>; **149**
 <operand2>; **149**
 <operand3>; **149**
 <operand4>; **149**
 <operand5>; **149**
 <operator application>; 154; **156**; 157
 <operator definition>; 18; 27; 138; 146; **147**; 148; 196
 <operator definitions>; 117; 118; 126; **146**
 <operator diagram>; 18; 27; 138; **147**; 148
 <operator heading>; 93; **147**
 <operator identifier>; 120; 122; 123; 126; 147; 149; **150**; 156
 <operator list>; **119**; 129; 136
 <operator name>; 19; **119**; 120; 123; 126; 137; 138; 141; 147
 <operator result>; **147**
 <operator signature>; 19; **119**; 120; 121; 135; 147; 150
 <operator text area>; 93; 147; **148**
 <operators>; 117; 118; **119**; 164; 182
 <option area>; 25; 32; 93; 98; **99**; 195
 <option outlet1>; **101**
 <option outlet2>; **101**
 <option symbol>; **99**
 <ordering>; 19; 119; 120; **132**; 199
 <other character>; **15**; 128
 <out-connector area>; 57; 59; **60**; 93; 197
 <out-connector symbol>; 20; **60**
 <outlet symbol>; **92**; 94; 95; 96
 <output area>; 59; **66**; 67; 93
 <output body>; **66**; 67; 195; 196; 197
 <output symbol>; 20; **66**
 <output>; 47; 58; **66**; 67; 88; 170; 171
 <overline>; **14**
 <package definition>; 18; **24**; 26; 194
 <package diagram>; 18; 24; **25**
 <package heading>; **25**
 <package list>; 23; **24**; 25; 26; 194; 195; 198
 <package name>; 24; 25; 26
 <package reference area>; **25**; 29; 93
 <package reference clause>; 19; **24**; 25; 26; 28; 195
 <package text area>; **25**; 93
 <package>; 23; **24**; 25; 26; 27; 195; 198
 <page number area>; 20; **21**
 <page number>; **21**
 <page>; **20**; 21
 <parameter kind>; 39; **40**; 42; 172; 179; 180
 <parameters of sort>; 19; **33**; 39
 <parent sort identifier>; 26; **133**; 134; 159
 <partial regular expression>; **143**; 144
 <partial type definition>; 18; 19; **117**; 118; 120; 121; 129; 132; 134; 138; 145; 153; 154; 159; 164; 199
 <path item>; **17**; 18; 26; 120
 <PId expression>; 66; 112; 114; 160; **161**; 162
 <plain input symbol>; **54**; 71
 <plain output symbol>; **66**; 71
 <predefined sort>; 97
 <primary>; **149**; 151; 152
 <priority input area>; 93; **104**
 <priority input association area>; 53; 93; **104**
 <priority input list>; 103; **104**
 <priority input symbol>; 20; **104**

<priority input>; 52; **104**; 105
 <procedure area>; **40**; 93; 99
 <procedure body>; 39; **40**; 42; 169; 197
 <procedure call area>; 59; **64**; 65; 93
 <procedure call body>; **64**
 <procedure call symbol>; **64**; 113
 <procedure call>; 41; 58; **64**; 163; 164
 <procedure constraint>; **179**
 <procedure context parameter>; 178; **179**
 <procedure definition>; 18; 24; 27; 28; 31; 33; 37; **39**; 40; 98; 118; 148; 196
 <procedure diagram>; 18; 27; 38; **40**; 41; 148; 167; 168; 169; 170
 <procedure formal parameter constraint>; **179**
 <procedure formal parameters>; **39**; 41; 42; 164
 <procedure graph area>; 40; **41**; 42; 57; 93
 <procedure heading>; 40; **41**
 <procedure identifier>; 39; 41; 64; 65; 164; 179; 180; 196
 <procedure name>; 33; 39; 40; 41; 179; 180
 <procedure preamble>; 33; **39**; 40; 41; 184
 <procedure result>; 18; 39; **40**; 41; 42; 61; 164; 195
 <procedure signature>; 112; 134; **179**; 180
 <procedure start area>; **41**; 42; 93
 <procedure start symbol>; **41**; 147
 <procedure symbol>; 40; **41**
 <procedure text area>; **40**; 93; 99
 <process area>; **31**; 32; 46; 93; 99
 <process body>; **33**; 34; 37
 <process constraint>; 17; **179**
 <process context parameter>; 17; 166; 167; 178; **179**
 <process definition>; 18; 27; 30; 32; **33**; 34; 37; 46; 47; 78; 98; 102; 103; 118; 174; 177; 196
 <process diagram>; 18; 27; 31; **34**; 35; 196
 <process graph area>; 34; **35**; 38; 57; 60; 93; 99
 <process heading>; 34; **35**
 <process identifier>; 17; 33; 35; 45; 63; 64; 66; 67; 174; 179; 196
 <process interaction area>; **31**; 93; 168
 <process name>; 17; 31; 33; 35; 173; 179
 <process signature>; **179**
 <process symbol>; **31**; 174; 176
 <process text area>; 34; **35**; 93; 99; 169
 <process type body>; 60; 168; **169**; 170
 <process type definition>; 18; 24; 27; 28; 30; 32; 46; 47; 63; 66; 98; 118; **168**; 169
 <process type diagram>; 18; 27; 46; 99; 167; 168; **169**
 <process type expression>; 173; 174
 <process type graph area>; 60; 93; 99; **169**
 <process type heading>; **169**
 <process type identifier>; 168; 169
 <process type name>; 168; 169
 <process type reference>; 93; 99; 167; 168; **169**
 <process type symbol>; **169**
 <properties expression>; **117**; 118; 126; 133; 139; 140; 141; 145
 <qualifier>; **17**; 18; 19; 27; 120; 126; 127; 140; 150; 152
 <quantification>; **122**; 123
 <quantified equations>; **122**; 123
 <question expression>; 68
 <question>; **68**; 69; 106; 108
 <quote>; **127**
 <quoted operator>; 17; 18; 19; 126; **127**; 150
 <range condition>; 68; 70; 133; 134; **135**; 195; 199
 <referenced definition>; 23; 24; **26**; 27; 194
 <regular element>; **143**; 144
 <regular expression>; **143**; 144
 <regular interval>; 128; **143**; 144
 <remote procedure call area>; 59; 93; **113**
 <remote procedure call body>; **112**; 113

<remote procedure call>; 58; **112**; 113; 114; 163
 <remote procedure context parameter>; 178; **180**
 <remote procedure definition>; 24; 28; 29; 31; 33; 35; 98; **112**; 113; 180
 <remote procedure identifier list>; 107; **112**; 113
 <remote procedure identifier>; 39; 40; 112; 113
 <remote procedure input area>; 54; 93; **113**; 114
 <remote procedure input transition>; 54; **112**; 113; 114
 <remote procedure name>; 37; 112; 113
 <remote procedure save area>; 55; 93; **113**; 114
 <remote procedure save>; 55; 107; **112**; 113; 114
 <remote variable context parameter>; 178; **181**
 <remote variable definition>; 24; 28; 29; 30; 33; 35; 98; **110**; 111; 181
 <remote variable identifier>; 50; 110
 <remote variable name>; 37; 110; 181
 <reset area>; 59; **71**; 93
 <reset statement>; **70**; 71
 <reset>; 58; **70**; 71; 195
 <restricted equation>; **124**
 <restriction>; **124**
 <result>; 19; 119; **120**; 121; 136; 147; 150
 <return area>; 42; 59; **61**; 93
 <return symbol>; **61**
 <return>; 42; 58; **61**; 199
 <reverse>; **89**; 180
 <right curly bracket>; **14**
 <right square bracket>; **14**
 <save area>; 53; **55**; 93
 <save association area>; **53**; 93
 <save list>; **55**; 103; 107
 <save part>; 52; **55**; 107; 114
 <save symbol>; 55; **56**; 113
 <scope unit kind>; **17**; 18; 117; 118
 <sdl specification>; **23**; 25; 26; 91; 194; 195
 <second constant>; 135
 <select definition>; 24; 28; 29; 30; 33; 35; 37; 38; 39; 40; **98**; 99; 147; 148; 195
 <service area>; **35**; 46; 93; 99
 <service body>; **37**
 <service definition>; 18; 27; 33; 34; **37**; 38; 47; 78; 98; 103; 118; 175; 196
 <service diagram>; 18; 27; 35; **38**; 196
 <service graph area>; **38**; 93; 170
 <service heading>; **38**
 <service identifier>; 37; 38; 45; 174
 <service interaction area>; 34; **35**; 93; 169
 <service name>; 33; 35; 37; 38; 174
 <service symbol>; **35**; 174; 176
 <service text area>; **38**; 93; 99; 170
 <service type body>; **170**
 <service type definition>; 18; 24; 27; 28; 31; 33; 98; 118; **170**; 171
 <service type diagram>; 18; 27; 99; 167; 168; 169; **170**
 <service type expression>; 174; 175
 <service type heading>; **170**
 <service type identifier>; 170
 <service type name>; 170
 <service type reference>; 93; 99; 167; 168; 169; **170**
 <service type symbol>; **170**
 <set area>; 59; **70**; 93
 <set statement>; **70**; 71
 <set>; 58; **70**; 71; 195
 <signal constraint>; **180**
 <signal context parameter>; 19; 178; **180**; 198
 <signal definition item>; **49**
 <signal definition>; 18; 19; 24; 28; 29; 30; 33; 35; **49**; 89; 98; 111; 113; 199
 <signal identifier>; 49; 54; 66; 67; 88; 103; 180

<signal list area>; 43; 44; 46; **49**; 93; 176
 <signal list definition>; 24; 28; 29; 30; 33; 35; **49**; 98
 <signal list identifier>; 26; 49
 <signal list item>; **49**
 <signal list name>; 49
 <signal list symbol>; 49; **50**
 <signal list>; 33; 43; 45; 47; **49**; 55; 88; 169; 170; 171; 175; 176
 <signal name>; 49; 111; 113; 180
 <signal refinement>; 49; **89**; 118; 180
 <signal route definition area>; 31; 35; **46**; 48; 93; 99
 <signal route definition>; 30; 32; 33; 34; **45**; 46; 47; 98
 <signal route endpoint>; **45**
 <signal route identifier>; 48; 66
 <signal route identifiers>; 34; **48**; 177
 <signal route name>; 45; 46
 <signal route path>; **45**; 177
 <signal route symbol 1>; **46**
 <signal route symbol 2>; **46**
 <signal route symbol>; 20; **46**; 92; 95; 176
 <signal route to route connection>; 33; 47; **48**; 98; 177; 196
 <signal signature>; **180**
 <simple expression>; **97**; 100; 101; 194; 195
 <solid association symbol>; 20; **22**; 53; 92; 95; 104; 105
 <sort constraint>; **182**
 <sort context parameter>; 18; 49; 118; 178; **182**; 198; 199
 <sort identifier>; 120; 180
 <sort list>; **49**; 70; 180; 181
 <sort name>; 17; 117; 118; 120; 182
 <sort signature>; **182**
 <sort type expression>; 137; 138
 <sort>; 17; 18; 26; 33; 40; 42; 49; 50; 51; 110; **120**; 123; 133; 136; 142; 143; 145; 147; 159; 163; 164; 179; 180; 181; 182
 <space>; 15; 16; 17; 128
 <special>; 14; **15**; 128
 <specialization>; 19; 39; 41; 49; 166; 167; 168; 169; 170; 172; 179; **182**; 197; 198; 199
 <spelling term>; 122; **145**; 146
 <spontaneous designator>; **56**
 <spontaneous transition area>; 53; **56**; 57; 93
 <spontaneous transition association area>; **53**; 93
 <spontaneous transition>; 52; **56**; 57; 107
 <start area>; 35; **51**; 52; 93; 169
 <start symbol>; **51**
 <start>; 33; 40; **51**; 52; 104; 147
 <state area>; 35; 41; **53**; 59; 93; 169
 <state list>; **52**; 53; 102
 <state name>; 19; 52; 53; 59; 102; 103; 104
 <state symbol>; **53**; 59
 <state>; 33; 40; **52**; 53; 102; 103; 104; 106; 108; 112; 114
 <stimulus>; **54**; 55; 103; 104; 105; 107; 197; 199
 <stop symbol>; 59; **61**; 93
 <stop>; 58; **61**
 <structure definition>; 125; **136**
 <structure primary>; 149; 150; **152**
 <sub expression>; **149**
 <subchannel identifiers>; **84**; 87; 177
 <subsignal definition>; **89**; 180
 <synonym constraint>; **181**
 <synonym context parameter>; 118; 178; **181**
 <synonym definition item>; **142**
 <synonym definition>; 97; **142**; 150; 153; 195
 <synonym identifier>; 147; 150
 <synonym name>; 142; 181
 <synonym>; 97; 149; **150**

<syntactical unit>; 20; **21**
 <syntype definition>; **133**; 134; 153; 154; 159; 195
 <syntype identifier>; 132; 133; 134; 180
 <syntype name>; 133
 <syntype>; 120; **132**; 133; 150
 <system definition>; 18; 23; 25; **26**; 27; 113; 118; 172; 195; 196; 198
 <system diagram>; 18; 26; **29**; 196
 <system heading>; **29**
 <system name>; 26; 28; 29; 172
 <system text area>; 29; 31; 85; 87; 93; 99; 166
 <system type definition>; 18; 24; 27; 98; 118; **166**; 167
 <system type diagram>; 18; 25; 27; 99; **166**
 <system type expression>; 172
 <system type heading>; 166; **167**
 <system type identifier>; 166; 167
 <system type name>; 166; 167
 <system type reference>; 25; 99; **167**
 <system type symbol>; **167**
 <task area>; 59; **62**; 93
 <task body>; 55; **62**; 195
 <task symbol>; **62**; 70; 71; 110
 <task>; 42; 55; 58; **62**
 <term>; **122**; 123; 125; 127; 130
 <terminator statement>; 57; **58**; 68; 69; 101
 <terminator>; **58**
 <text extension area>; **22**; 94
 <text extension symbol>; 20; **22**
 <text symbol>; **22**; 25; 29; 35; 38; 40; 148
 <text>; **15**; 21; 22; 164
 <textual block reference>; **28**; 98
 <textual block substructure reference>; 30; **84**; 167
 <textual block type reference>; 24; 28; 31; 98; **167**
 <textual channel substructure reference>; 43; **87**
 <textual endpoint constraint>; **175**; 176
 <textual operator reference>; 146; **147**; 148
 <textual procedure reference>; 24; 28; 31; **33**; 37; 40; 98
 <textual process reference>; 30; **31**; 34; 98
 <textual process type reference>; 24; 28; 30; 98; **169**
 <textual service reference>; 33; 34; 98
 <textual service type reference>; 24; 28; 31; **33**; 98; **170**
 <textual system definition>; 18; 26; **28**
 <textual system type reference>; 24; 98; **166**
 <textual typebased block definition>; 28; 32; 34; 43; 44; 84; 85; 98; **173**; 176; 196
 <textual typebased process definition>; 30; 32; 34; 45; 47; 98; **173**; 174; 176; 196
 <textual typebased service definition>; 32; 33; 34; 37; 45; 47; 98; **174**; 175; 196
 <textual typebased system definition>; 28; **172**; 196
 <Time expression>; 70; 71
 <timer active expression>; 160; **162**; 163; 198
 <timer constraint>; **181**
 <timer context parameter>; 166; 167; 169; 170; 178; **181**
 <timer definition item>; **70**
 <timer definition>; 33; 35; 37; 38; **70**; 98
 <timer identifier>; 49; 54; 70; 162
 <timer name>; 70; 181
 <transition area>; 41; 51; 54; 56; 57; **59**; 60; 69; 94; 101; 104; 105; 113; 147
 <transition option area>; 59; 94; **101**; 195
 <transition option symbol>; **101**
 <transition option>; 32; 58; 68; **100**; 101; 102; 195
 <transition string area>; **59**; 94
 <transition string>; 57; **58**; 60; 68; 69; 101
 <transition>; 51; 54; 55; 56; 57; **58**; 68; 69; 101; 104; 105; 106; 112; 113; 114; 147; 195
 <type expression>; **171**; 172; 178; 179; 182; 183; 199
 <type in block area>; 31; 94; **168**

<type in process area>; 34; 94; **169**
 <type in system area>; 25; 29; 85; 87; 94; 166; **167**
 <typebased block heading>; **173**
 <typebased process heading>; **173**; 174
 <typebased service heading>; **174**
 <typebased system heading>; **172**
 <underline>; **14**; 15; 16; 17; 18; 128; 194
 <unquantified equation>; **122**; 123; 124
 <upward arrow head>; **14**
 <valid input signal set>; **33**; 34; 35; 36; 37; 38; 46; 47; 168; 169; 170; 171
 <value identifier>; 19; 122; 145; 146
 <value name>; 122; 123; 145
 <value returning procedure call>; 42; 148; 149; **163**; 164; 196; 198
 <variable access>; 154; **155**
 <variable context parameter>; 166; 167; 169; 178; **181**
 <variable definition>; 17; 33; 35; 37; 38; 40; **50**; 51; 98; 111; 147; 148; 170
 <variable identifier>; 55; 64; 110; 155; 157
 <variable name>; 33; 40; 42; 50; 147; 181
 <variable>; 54; 55; **157**; 158; 159
 <variables of sort>; **50**
 <vertical line>; **14**
 <via path element>; **66**; 67
 <via path>; **66**; 67; 88
 <view definition>; 17; 33; 35; 37; 38; 50; **51**; 98
 <view expression>; 160; **162**; 198
 <view identifier>; 162
 <view name>; 51
 <virtuality constraint>; 39; 41; 167; 168; 169; 170; **184**
 <virtuality>; 21; 39; 41; 42; 51; 52; 54; 55; 56; 57; 104; 105; 106; 112; 113; 114; 147; 167; 168; 169; 170; **183**; 184; 185; 197
 <word>; **14**; 17; 164
 abstract data types (G); 115
 action (G); 59
active; 15; 162
 active expression (G); 148; 154
 active timer (G); 71; 163
Active-expression; 148; **154**
 actual context parameters (G); 177
 actual parameter (G); 63; 64; 91
adding; 15; 136; 137; 139; 141; 175; 176; 182; 188; 193
all; 15; 66; 67; 122; 124; 129; 132; 137; 138; 139; 145; 146; 197
alternative; 15; 100; 102; 130; 164; 165
Alternative-expression; **155**
and; 15; 48; 74; 78; 84; 86; 87; 88; 100; 127; 129; 132; 135; 137; 139; 149; 190
And-operator-identifier; **134**
any; 15; 68; 69; 163
 anyvalue expression (G); 163
Anyvalue-expression; 160; **163**
Argument-list; **119**
as; 15; 39; 40; 50; 110; 113
 assignment statement (G); 157
Assignment-statement; 62; **157**
 association area (G); 53; 87
atleast; 15; 139; 172; 175; 178; 179; 180; 182; 184; 187; 189; 193
 axiom (G); 116; 121
axioms; 15; 117; 137; 139; 140; 141; 142; 145; 146
 basic SDL (G); 13
block; 15; 17; 20; 24; 28; 30; 31; 73; 74; 86; 88; 96; 100; 167; 168; 173; 187; 188; 189; 190
 block (G); 32
 block substructure (G); 84
 block tree diagram (G); 84
 block type (G); 168
Block-definition; 28; **30**; 83

Block-identifier; **42**
Block-name; 17; **30**
Block-qualifier; 16; **17**
Block-substructure-definition; 30; **83**
Block-substructure-name; 17; **83**
Block-substructure-qualifier; 16; **17**
Boolean-expression; **155**
call; 15; 64; 77; 112; 114; 189; 193
Call-node; 58; **64**
channel; 15; 43; 73; 86; 88; 96; 100; 189
channel (G); 44
channel substructure (G); 88
Channel-connection; **83**
Channel-definition; 28; **42**; 83
Channel-identifier; 47; 65; **83**
Channel-name; **42**
Channel-path; **42**
Channel-to-route-connection; 30; **47**
Closed-range; **134**
comment; 15; 21; 72
comment (G); 21
communication path (G); 65
complete valid input signal set (G); 36; 38
Composite-term; **121**
Condition; **130**
Condition-item; **134**
conditional expression (G); 155
Conditional-composite-term; 121; **130**
Conditional-equation; 121; **124**
Conditional-expression; 154; **155**
Conditional-ground-term; 121; **130**
Conditional-term; **130**
connect; 15; 48; 74; 78; 84; 86; 87; 88; 190
connect (G); 47
connection; 15; 57
connector (G); 57
Consequence; **130**
Consequence-expression; **155**
consistent partitioning subset (G); 82
consistent refinement subset (G); 90
constant; 15; 126; 139; 140; 141
constants; 15; 70; 133; 134
constraint (G); 176; 177; 182
context parameter (G); 177
continuous signal (G); 105
create; 15; 63
create (G); 63
create line area (G); 31
create request (G); 63
Create-request-node; 58; **63**
dash nextstate (G); 104
data type (G); 115
data type definition (G); 116
Data-type-definition; 28; 30; 32; 37; 39; 83; **117**
dcl; 15; 50; 75; 76; 77; 79; 181; 192; 193
decision; 15; 68; 76; 105
decision (G); 69
Decision-answer; **68**
Decision-node; 58; **68**
Decision-question; **68**
default; 15; 159
default initialization (G); 159
Destination; **45**

Destination-block; **42**
 diagram (G); 27
Direct-via; **65**
else; 15; 68; 69; 101; 102; 130; 131; 140
Else-answer; **68**
 enabling condition (G); 106
endalternative; 15; 100; 102; 164; 165
endblock; 15; 20; 30; 74; 167; 187; 188; 190
endchannel; 15; 43; 73; 86; 88; 96; 100; 189
endconnection; 15; 57
enddecision; 15; 68; 76; 105
endgenerator; 15; 139; 140
endmacro; 15; 91
endnewtype; 15; 117; 133; 134; 139; 145; 146; 165; 182; 193
endoperator; 15; 147
endpackage; 15; 24
 endpoint constraint (G); 175
endprocedure; 15; 39; 76; 189; 193
endprocess; 15; 33; 75; 77; 78; 168; 189; 192; 193
endrefinement; 15; 89
endselect; 15; 98; 100
endservice; 15; 37; 79; 80; 170
endstate; 15; 52; 79; 80
endsubstructure; 15; 84; 86; 87; 88
endsyntype; 15; 70; 133; 134
endsystem; 15; 20; 28; 73; 100; 166; 189
 entity kind (G); 19
env; 15; 43; 45; 73; 74; 78; 86; 87; 88; 100; 176; 177; 187; 188; 190
 environment (G); 13
Equation; 117; **121**
 equation (G); 116; 121
Equations; **117**; 121
error; 15; 131; 140; 148; 154; 155
 error (G); 131
Error-term; 121; **131**; 154
export; 15; 110; 112
 export (G); 110
 export operation (G); 111
exported; 15; 21; 39; 40; 41; 50; 110; 111; 113
 exported variable (G); 50
 exporter (G); 111
Expression; 63; 64; 65; 68; 70; **148**; 155; 156; 157; 162
 expression (G); 23; 148
external; 15; 97; 100
 external synonym (G); 97
External-signal-route-identifier; **47**
 Extract! (G); 136
fi; 15; 130; 131; 140
 field (G); 136
finalized; 15; 183; 184; 185; 189
 flow line (G); 20; 60
for; 15; 122; 124; 129; 132; 139; 145; 146
 formal context parameter (G); 177
 formal parameter (G); 36; 41; 91
fpar; 15; 33; 39; 76; 78; 91; 179; 189; 192; 193
from; 15; 43; 45; 73; 74; 78; 86; 88; 96; 100; 175; 187; 188; 189; 190
gate; 15; 175; 187; 192; 193
 gate (G); 175
 gate constraint (G); 175
generator; 15; 25; 139; 140
 generator (G); 139
 graph (G); 32; 37; 39
Graph-node; **58**

graphical gate constraint (G); 176
 ground expression (G); 148; 149
Ground-expression; 50; 134; 148; **149**
Ground-term; **121**; 149
 hierarchical structure (G); 82
 Identifier; **16**; 42; 45; 47; 53; 64; 70; 83; 119; 121; 132; 134
 identifier (G); 17
if; 15; 98; 99; 100; 131; 140
 imperative operator (G); 160
Imperative-operator; 154; **160**
 implicit transition (G); 103
import; 15; 110
 import (G); 111
 import expression (G); 110
 import operation (G); 111
imported; 15; 110; 112
 imported variable (G); 110
 importer (G); 111
in; 15; 40; 41; 42; 64; 76; 113; 114; 122; 124; 127; 129; 132; 133; 134; 139; 145; 146; 149; 164; 172; 175; 176; 180;
 187; 192; 193; 196
 in variable (G); 41
 in-connector (G); 57
In-parameter; **39**
 in/out variable (G); 41
 infix operator (G); 127
 informal text (G); 20
Informal-text; **20**; 62; 68; 121
 inherit (G); 138; 182
inherits; 15; 137; 139; 182; 188; 189; 193
 inlet (G); 95
Inout-parameter; **39**
input; 15; 54; 56; 75; 76; 77; 79; 80; 104; 105; 106; 108; 111; 112; 114; 192; 193
 input (G); 54
 input port (G); 36
Input-node; 52; **53**
interface; 15; 25
join; 15; 60; 76; 106; 107; 108
 join (G); 60
 keyword (G); 15
 label (G); 57
 level (G); 82
 lexical rules (G); 13
 lexical unit (G); 13
literal; 15; 126; 139; 140; 141
 literal (G); 23; 115; 120
Literal-operator-identifier; **121**
Literal-operator-name; **119**
Literal-signature; **119**
literals; 15; 117; 119; 120; 121; 128; 129; 132; 137; 139; 140; 141; 142; 145; 146; 165
macro; 15; 95; 96
 macro (G); 91
 macro call (G); 95
macrodefinition; 15; 91; 92
macroid; 15; 91; 94
 Make! (G); 136
map; 15; 144; 145; 146
 merge area (G); 60
mod; 15; 127; 149
 Modify! (G); 136
Name; 16; **17**; 28; 30; 32; 37; 39; 42; 45; 48; 50; 52; 70; 83; 117; 119; 121; 132
 name (G); 17
nameclass; 15; 143; 144
newtype; 15; 17; 24; 25; 115; 117; 128; 133; 134; 138; 139; 145; 146; 165; 182; 193; 195

newtype (G); 115
nextstate; 15; 59; 75; 76; 79; 80; 102; 105; 106; 192; 193
 nextstate (G); 60
Nextstate-node; 58; **59**
nodelay; 15; 43; 110; 112
noequality; 15; 129; 136; 137; 138; 182
none; 15; 56; 75; 80; 192
not; 15; 127; 129; 139; 149
 note (G); 15
now; 15; 71; 160
 now expression (G); 160
Now-expression; **160**
Number-of-instances; **32**
offspring; 15; 36; 37; 63; 161
 offspring (G); 36
Offspring-expression; **161**
Open-range; **134**
operator; 15; 17; 126; 139; 140; 141; 147
 operator (G); 115; 116; 118; 120
 operator signature (G); 120
Operator-application; 154; **156**
Operator-identifier; **121**; 134; 156
Operator-name; **119**
Operator-signature; **119**
operators; 16; 117; 119; 120; 121; 128; 132; 137; 139; 140; 141; 142; 145; 146; 165; 193
 option (G); 98; 100
or; 16; 127; 132; 135; 139; 143; 144; 149
Or-operator-identifier; **134**
ordering; 16; 132; 182
 ordering operators(G); 132
Origin; **45**
Originating-block; **42**
out; 16; 40; 42; 64; 76; 113; 114; 134; 164; 172; 175; 176; 180; 187; 192; 196
 out-connector (G); 60
 outlet (G); 92
output; 16; 66; 75; 76; 79; 80; 105; 106; 107; 111; 112; 114; 192
 output (G); 65
Output-node; 58; **65**
package; 16; 17; 24; 25; 26; 70; 129; 130
 package (G); 24
 package interface (G); 25
 package reference (G); 24
 page (G); 20
parent; 16; 36; 63; 161
 parent (G); 36
Parent-expression; **161**
Parent-sort-identifier; **132**
 partial type definition (G); 117
 partitioning (G); 82
Path-item; **16**
Pid-expression; 160; **161**
 predefined (G); 128
 predefined data (G); 115; 128
priority; 16; 79; 104; 105
 priority input (G); 104
procedure; 16; 17; 24; 33; 39; 41; 76; 112; 113; 179; 180; 189; 193
 procedure (G); 39
 procedure call (G); 41; 65
 procedure constraint (G); 179
 procedure context parameter (G); 179
 procedure graph (G); 41
 procedure signature (G); 179
 procedure start (G); 41

Procedure-definition; 32; 37; **39**
Procedure-formal-parameter; **39**
Procedure-graph; **39**
Procedure-identifier; **64**
Procedure-name; 17; **39**
Procedure-qualifier; 16; **17**
Procedure-start-node; **39**
process; 16; 17; 24; 31; 33; 35; 74; 77; 78; 168; 169; 173; 179; 187; 188; 189; 190; 192; 193
process (G); 32
process constraint (G); 179
process context parameter (G); 179
process graph (G); 32
process instance (G); 35
process signature (G); 179
process type (G); 169
Process-definition; 30; **32**
Process-formal-parameter; **32**
Process-graph; **32**
Process-identifier; **45**; 63; 65
Process-name; 17; **32**
Process-qualifier; 16; **17**
Process-start-node; 32; **51**
provided; 16; 105; 107
Qualifier; **16**
qualifier (G); 17
Quantified-equations; **121**
Range-condition; 68; 132; **134**
redefined; 16; 183; 184; 185; 189
referenced; 16; 28; 31; 33; 73; 74; 78; 84; 86; 87; 88; 96; 100; 147; 166; 167; 169; 170; 187; 188; 189; 190
referenced definition (G); 27
refinement; 16; 89
refinement (G); 89
rem; 16; 127; 149
remote; 16; 25; 110; 112; 180; 181
remote procedure call (G); 112
remote procedure definition (G); 112
remote procedure input transition (G); 112
remote procedure save (G); 112
remote variable definition (G); 110
reset; 16; 70
reset (G); 70
Reset-node; 58; **70**
Restricted-equation; **124**
Restriction; **124**
Result; **119**
retained signal (G); 36
return; 16; 61; 193
return (G); 41; 61
Return-node; 58; **61**
returns; 16; 40; 147; 179
revealed; 16; 40; 50; 170
revealed attribute (G); 50
reverse; 16; 89; 90
save; 16; 55; 112; 114
save (G); 55
Save-signalset; 52; **55**
scope unit (G); 19
select; 16; 98; 99; 100
selection (G); 97
self; 16; 36; 56; 63; 71; 80; 105; 106; 107; 161
self (G); 36
Self-expression; **161**
sender; 16; 36; 37; 55; 56; 71; 76; 112; 114; 161

sender (G); 36
Sender-expression; **161**
service; 16; 17; 24; 33; 37; 38; 78; 79; 80; 170; 174
 service (G); 37
 service type (G); 171
Service-decomposition; **32**
 Service-definition; 32; **37**
Service-graph; **37**
Service-identifier; **45**
Service-name; 17; **37**
Service-qualifier; 16; **17**
Service-start-node; **37**
set; 16; 70
 set (G); 70
Set-node; 58; **70**
signal; 16; 17; 24; 49; 73; 74; 76; 77; 78; 86; 88; 100; 180; 189; 190; 193
 signal (G); 49
 signal constraint (G); 180
 signal context parameter (G); 180
 signal definition (G); 49
 signal list (G); 49
 signal route (G); 46
 signal signature (G); 180
Signal-definition; 28; 30; 32; **48**; 83; 89
Signal-destination; **65**
Signal-identifier; **42**; 45; 53; 55; 65
Signal-name; 17; **48**
Signal-qualifier; 16; **17**
Signal-refinement; 48; **89**
Signal-route-definition; 30; 32; **44**
Signal-route-identifier; **47**; 65
Signal-route-name; 44; **45**
Signal-route-path; 44; **45**
Signal-route-to-route-connection; 32; **47**
signallist; 16; 24; 49
signalroute; 16; 45; 74; 78; 187; 188; 190
signalset; 16; 33
Signature; 117; **119**
 signature (G); 118; 120
 simple expression (G); 98
 sort (G); 115
 sort constraint (G); 182
 sort context parameter (G); 182
 sort signature (G); 182
Sort-identifier; **119**; 121; 132
Sort-name; 17; **117**
Sort-qualifier; 16; **17**
Sort-reference-identifier; 32; 39; 48; 50; 51; **119**; 163
Sorts; **117**
 specialization (G); 182
spelling; 16; 145; 146
 spontaneous transition (G); 56
Spontaneous-transition; 52; **56**
start; 16; 51; 75; 76; 79; 80; 192; 193
 start (G); 51
state; 16; 52; 75; 76; 79; 80; 192; 193
 state (G); 52
State-name; **52**; 59
State-node; 32; 37; 39; **52**
stop; 16; 61; 75; 79; 192
 stop (G); 61
Stop-node; 58; **60**
struct; 16; 115; 136; 137

structure sort (G); 136
Sub-block-definition; **83**
Sub-channel-identifier; **83**
 subblock (G); 82; 83
 subchannel (G); 83
 subsignal (G); 89
Subsignal-definition; **89**
substructure; 16; 17; 84; 85; 86; 87; 88
 subtype (G); 166; 182
 supertype (G); 182
synonym; 16; 25; 97; 100; 142; 181
 synonym (G); 142
 synonym context parameter (G); 181
syntype; 16; 70; 133; 134
 syntype (G); 132
Syntype-definition; 28; 30; 32; 37; 39; 83; **132**
Syntype-identifier; 119; **132**
Syntype-name; **132**
system; 16; 17; 20; 24; 26; 28; 29; 73; 166; 167; 172; 189
 system (G); 13
 system type (G); 166
System-definition; **28**
System-name; 17; **28**
System-qualifier; **16**
task; 16; 62; 72; 75; 76; 79; 102; 105; 106; 107; 112; 192; 193
 task (G); 62
Task-node; 58; **62**
Term; **121**; 130
 term (G); 116; 117; 123
Terminator; **58**
 text extension symbol (G); 22
 textual endpoint constraint (G); 175
then; 16; 130; 131; 140
this; 16; 63; 64; 65; 66; 67; 172; 196
Time-expression; **70**
timer; 16; 70; 181
 timer (G); 71; 163
 timer active expression (G); 163
 timer context parameter (G); 181
Timer-active-expression; 160; **162**
Timer-definition; 32; 37; **70**
Timer-identifier; **70**; 162
Timer-name; **70**
to; 16; 43; 45; 67; 73; 74; 75; 78; 79; 80; 86; 88; 96; 100; 105; 106; 107; 111; 112; 114; 175; 187; 188; 189; 190; 192
Token; **17**
Transition; 37; 39; 51; 53; 56; **58**; 68
 transition (G); 59
 transition string (G); 58
type; 16; 17; 24; 120; 130; 139; 140; 141; 166; 167; 168; 169; 170; 187; 188; 189; 190; 192; 193
 type expression (G); 171
Unquantified-equation; **121**; 124
use; 16; 24; 26; 185
 valid input signal set (G); 36
 value (G); 23; 116
 value returning procedure (G); 163
Value-identifier; **121**
Value-name; **121**
 variable (G); 23
 variable context parameter (G); 181
 variable definition (G); 50
Variable-access; 154; **155**
Variable-definition; 32; 37; 39; **50**

Variable-identifier; **53**; 155; 157
Variable-name; 32; 39; **50**
via; 16; 43; 45; 66; 67; 88; 187; 188; 189; 190; 196; 197
view; 16; 162
view (G); 51
view definition (G); 51
view expression (G); 162
View-definition; 32; 37; **51**
View-expression; 160; **162**
View-identifier; **162**
View-name; **51**
viewed; 16; 51
virtual; 16; 183; 184; 189
virtual continuous signal (G); 106
virtual input (G); 55; 185
virtual priority input (G); 104; 185
virtual procedure start (G); 42
virtual remote procedure input (G); 114
virtual save (G); 56
virtual spontaneous transition (G); 57
virtual start (G); 52; 184
virtual type (G); 183
virtuality (G); 183
virtuality constraint (G); 184
visibility (G); 19
with; 16; 43; 45; 73; 74; 78; 86; 88; 96; 100; 175; 187; 188; 189; 190; 192; 193
xor; 16; 127; 149

Annex B

SDL Glossary

(This annex forms an integral part of this Recommendation)

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, informal 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 *italicized* in a definition may also be found in the glossary. If an italicized phrase, e.g. *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*. SDL keywords are in **bold**.

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

Abstract data types define data in terms of abstract properties rather than in terms of a concrete implementation. An *abstract data type* is a class of type which defines 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, 5.1]

abstract grammar

The *abstract grammar* defines the *semantics of SDL*. The *abstract grammar* is described by the *abstract syntax* and the *well-formedness rules*. [1.2, 1.4.1]

abstract syntax

The *abstract syntax* is the means to describe the conceptual structure of an *SDL specification*. The *concrete syntaxes* are mapped to the *abstract syntax* to ensure that *SDL/PR* and *SDL/GR* are equivalent. [1.2]

action

An *action* is an operation which is executed within a *transition string*, e.g., a *task*, *output*, *decision*, *create request*, *set*, *reset*, *export*, or *procedure call*. [2.6.8.1, 2.7]

active expression

An *active expression* is an *expression* where the *value* depends on the current system state. An *active expression* accesses a *variable* or contains an *imperative operator*. [5.3.3.1, 5.4.2.1]

active timer

An *active timer* is a *timer* which has a *timer signal* in the *input port* of the owning *process* or is scheduled to produce a *timer signal* at some future time. [2.8, 5.4.4.5]

actual context parameter

An *actual context parameter* is an *identifier* denoting the actual definition for a corresponding *formal context parameter*. The *actual parameter identifier* may either be an *identifier* of a *formal context parameter* of a *subtype definition*, or denote a *definition* that is *visible* in the enclosing scope of the *type expression* or *visible* through a *package reference*. [6.2]

actual parameter

An *actual parameter* is an *expression* interpreted by 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]

anyvalue expression

An *anyvalue expression* is an *expression* which yields a non-specified *value* of the designated *sort* or *syntype*. [5.4.4.6]

area

An *area* is a two-dimensional region in the *concrete graphical syntax*. *Areas* 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*. [2.4.2.6]

array

Array is the *predefined data generator* used to introduce the concept of arrays, simplifying the definition of arrays. [Annex D]

assignment statement

An *assignment statement* is a statement in a task, which, when interpreted, associates a *value* to a *variable* replacing the previous *value* associated with the *variable*. [5.4.3]

association area

An *association area* is a connection between *areas* in an *interaction diagram* by means of an *association symbol*. There are five association areas: *channel substructure association area*, *input association area*, *priority input association area*, *continuous signal association area* and *save association area*. [1.5.3, 2.6.3, 3.2.3, 4.10.2, 4.11]

axiom

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, 5.2.3]

basic SDL

Basic SDL is the subset of *SDL* defined in § 2 of Recommendation Z.100. [2]

behaviour

In *SDL*, *behaviour* is either 1) externally observable behaviour: the set of sequences of responses of a *system* to sequences of stimuli, or 2) internally observable behaviour: a set of actions and tasks, triggered by a stimulus, executed before the state machine transitions to the next state. [1.1.3]

binding

Binding associates *actual parameters* with *formal parameters* (in a creation of a *process* or in a *procedure call*) or *actual context parameters* with (formal) *context parameters*. [1.3.1]

block

A *block* is part of a *system* or *block* and is the container for one or more process or one *block substructure*. A *block* is a *scope unit* and provides a static interface. When used by itself, *block* is a synonym for a *block instance*. [2.4.3]

block substructure

A *block substructure* is the *partitioning* of the *block* into *subblocks* and new *channels* at a lower *level of abstraction*. [3.2.2]

block tree diagram

A *block tree diagram* is an auxiliary document in *SDL/GR* representing the *partitioning* of a *system* into *blocks* at successively lower *levels of abstraction* by means of an inverted tree diagram (i.e., *block* at the top). [3.2.1]

block type

A *block type* is the association of a *name* and a set of properties that all *block instances* of this *type* will have. [6.1.1.2]

BNF (Backus-Naur Form)

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

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. [5.3.1.3, Annex D]

channel

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 substructure

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]

character

Character is a *predefined data sort* for which the *values* are the elements of the CCITT No. 5 alphabet, (e.g., 1, A, B, C, etc.). For the *character sort* the *ordering operators* are predefined. [Annex D]

charstring

Charstring is a *predefined data sort* for which the *values* are *strings* of *characters* and the *operators* are those of the *string predefined generator* instantiated for *characters*. [Annex D]

comment

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

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

A *communication path* is a transportation means that carries *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

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

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

The *concrete graphical grammar* is the *concrete grammar* for the graphical of *SDL/GR*. [1.2, 1.5.3]

concrete graphical syntax

The *concrete graphical syntax* is the *concrete syntax* for the graphical 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

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

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

A *conditional expression* is an *expression* containing a *Boolean expression* which controls whether the consequence *expression* or the alternative *expression* is interpreted. [5.4.2.3]

connect

Connect indicates the connection of *channels* to one or more *signal routes* or the interconnection of *signal routes*. [2.5.3]

connector

A *connector* in *SDL/PR* is the *label* on an *action*. 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*. [2.6.7, 2.6.8.2.2]

consistent partitioning subset

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

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]

constraint

The *constraint* of a *context parameter* constrains *actual context parameters* and defines the properties of the parameter known by the *parameterized type*. The *constraint* of a *virtual type* constrains the redefinitions and defines the properties of the *virtual type* known by the enclosing *type*. See also *gate constraint*. [6.2]

context parameter

A (formal) *context parameter* of a *parameterized type definition* is the association of an *identifier* and a *constraint*. A new *type* is defined (by a *type expression*) by associating some or all of the *context parameters* with actual *definitions*. [6.2]

continuous signal

A *continuous signal* is a *shorthand notation* which allows initiating a *transition* when the associated *Boolean* condition becomes True. [4.11]

control flow diagram

A *control flow diagram* is either a *process diagram*, a *procedure diagram*, or a *service diagram*.

create

Create is a synonym for *create request*.

create line area

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]

create request

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 definition*. [2.7.2]

dash nextstate

A *dash nextstate* is a *shorthand notation* indicating that the *nextstate* of the *process instance* is the current *state*. [4..9]

data type

Data type is a synonym for *abstract data type*.

data type definition

The *data type definition* at any given point in an *SDL specification* defines the validity of *operators*, *sorts*, and *expressions* and relationship between *expressions*. [5.2.1]

decision

A *decision* is an *action* within a *transition* which asks a question to which the answer can be obtained at that instant and, based on the answer selects one of the outgoing *transitions* from the *decision* to continue interpretation. [2.7.5]

default initialization

A *default initialization* is a notation for associating the same *value* to all *variables* of the specified *sort* before their associated *process* or *procedure* is interpreted. [5.4.3.3]

definition

A *definition* associates a name and a set of properties with a *type* or an *instance*. [1.3.1]

description

A *description* of a *system* is the description of its actual *behaviour*. [1.1]

diagram

A *diagram* is the *SDL/GR* representation for a part of a *specification*. [2.4.2]

duration

Duration is a *predefined data sort* for which the *values* are denoted as *reals* and represent the interval between two time instants. [Annex D]

enabling condition

An *enabling condition* is a means for conditionally accepting a *signal* for *input*. [4.12]

endpoint constraint

An *endpoint constraint* is the *SDL/GR* representation of a *constraint* on a *gate* constraining which *blocks/processes/services* may be at the other end of a *channel/signal route/service signal route* connected to the *gate*. [6.1.4]

entity kind

An *entity kind* is a categorization of *SDL types* based on similarity of use. [2.2.2]

environment

The term *environment* is a synonym for the *environment of a system*. When context allows, it may be a synonym for the *environment* of a *block*, *process*, *procedure* or a *service*.

environment of a system

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

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]

error

An *error* occurs during the interpretation of a *valid specification* of a *system* when one of the dynamic conditions of *SDL* is violated. Once an *error* has occurred, the subsequent *behaviour* of the *system* is not defined. [1.3.3]

export

The term *export* is a synonym for *export operation*.

exported variable

An *exported variable* is a *variable* which can be used in an *export operation*. Its *definition* contains the keyword **exported**. [2.6.1.1]

exporter

An *exporter* of a *variable* is the *process instance* which owns the *variable* and *exports its values*. [4.13]

export operation

An *export operation* is the execution of an *export action* by which the *exporter* discloses the current *value* of a *variable*. See also *import operation*. [4.13]

expression

An *expression* is either a *literal*, an *operator* application, a *synonym*, a *variable access*, a *conditional expression*, or an *imperative 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, 5.3.3.1]

external synonym

An *external synonym* of a *predefined data sort* whose *value* is not specified in the *system specification*. [4.3.1]

Extract!

Extract! is an implied *operator* for extracting a *field* from a *structured sort* and is implied in an *expression* when a *variable* is immediately followed by bracketed *expression(s)*. [5.3.1.10, 5.3.3.4, 5.3.3.5]

field

A *field* is an element of a *structured sort*. [5.3.1.10]

flow line

A *flow line* is a *symbol* used to connect *areas* in a *control flow diagram*. [2.2.4, 2.6.8.2.2]

formal context parameter

See *context parameter*.

formal parameter

A *formal parameter* is a *variable* name to which *actual values* are assigned or which are replaced by *actual variables*. [2.4.4, 2.4.6, 4.2]

gate

A *gate* is defined in a *block/process/service type* and represents a connection point for *channels/signal routes* connecting instances of the *type* with other instances or with the enclosing frame symbol. [6.1.4]

gate constraint

A *gate constraint* constrains the way *channels/signal routes* may be connected to the *gate*, and places restrictions on their incoming and outgoing *signals*. [6.1.4]

general parameters

The general *parameters* in both a *specification* and a *description* of a *system* relate to such *non-behavioural* matters as temperature limits, construction, exchange capacity, grade of service, etc., and are not defined in *SDL*. [1.1]

generator

A generator allows a parameterized text template to be defined which is expanded by transformation before the *semantics* of the *data types* are considered. [5.3.1.12, 7.2]

graph

A *graph* in the *abstract syntax* is a part of an *SDL specification* such as a *procedure graph*, *service graph*, or a *process graph*. [2.4.4, 2.4.5, 2.4.6]

graphical gate constraint

A *graphical gate constraint* is the *SDL/GR* representation of a *gate constraint*. [6.1.4]

ground expression

A *ground expression* is an *expression* containing only *operators*, *synonyms* and *literals*. [5.3.3.2]

hierarchical structure

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*. See also *block tree diagram*. [3.1]

identifier

An *identifier* is the unique identification of an object, formed from a *qualifier* part and a *name*. [2.2.2]

imperative operator

An *imperative operator* is a *view expression*, *timer active expression*, *import expression*, *anyvalue expression*, *now expression*, or one of the *PId* expressions: **self**, **parent**, **offspring**, or **sender**. [5.4.4]

implicit transition

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*. [4.8]

import

The term *import* is a synonym for *import operation*. [4.13]

imported variable

An *imported variable* is a *variable* used in an *import expression*. [4.13]

importer

An *importer* of an *imported variable* is the *process instance* which *imports* the *value*. [4.13]

import expression

An *import expression* specifies the keyword **import** and the *identifier* of a *variable*. [4.13]

import operation

An *import operation* is the execution of an *import expression* by which the *importer* accesses the *value* of an *exported variable*. [4.13]

inherit

A *specialized type* or *sort* has or *inherits* all the properties of its *supertype* or referenced *sort*. [5.3.1.11, 6.3]

in parameter

An *in parameter* is a *formal parameter* attribute where a *value* is passed to a *procedure* via an *actual parameter*. [2.4.6]

in/out parameter

An **in/out** parameter is a *formal parameter* attribute where a *formal parameter name* is used as a synonym for the *variable* (i.e., the *actual parameter* must be a *variable*). [2.4.6]

in-connector

See *connector*.

infix operator

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. [5.3.1.1]

informal text

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]

inlet

An *inlet* represents a line, such as a *channel* or a *flow line*, entering an *SDL/GR macro call*. [4.2.3]

input

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 port

The *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

An *instance* is a part of a system. An *instance* has properties and may exhibit *behaviour*. An *instance* of a *type* is an object which has all the properties of the type as given in the *type definition*. [1.3.1]

instantiation

Instantiation is the creation of an *instance* of a *type*. [1.3.1]

integer

Integer is a *predefined data sort* for which the *values* are the mathematical integers (... , -2, -1, 0, +1, +2, ...). For the *sort integer* the predefined operators *+*, *-*, ***, */*, and the ordering operators. [Annex D]

interaction diagram

An *interaction diagram* is a *block diagram*, *system diagram*, *channel substructure diagram*, or *block substructure diagram*.

join

A *join* indicates a change in the execution flow of a *transition string* by indicating the *connector* to or *label* of the *action* which should be executed next. [2.6.8.2.2]

keyword

A *keyword* is a reserved *lexical unit* in the *concrete textual syntax*. [2.2.1]

label

A *label* is a *name* followed by a colon and is used in the *concrete textual syntax* to indicate the target for transfer of control from a *join* which references the same *identifier* as the *label*. [2.6.7, 2.6.8.2.2]

level

The term *level* is a synonym for *level of abstraction*.

level of abstraction

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

Lexical rules are rules which define how *lexical units* are built from characters. [2.2.1, 4.2.1]

lexical unit

Lexical units are the terminal *symbols* of the *concrete textual syntax*. [2.2.1]

literal

A *literal* denotes a value; it is an operator without an argument. [2.3.3, 5.1.2, 5.2.2, 5.3.1.14]

macro

A *macro* 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

A *macro call* is an indication of a place where the *macro definition* with the same name should be expanded. [4.2.3]

Make!

Make! is an operation only used in *data type* definitions to form a *value* of a complex type (i.e., *structured sort*). [5.3.1.10]

merge area

A *merge area* is where one *flow line* connects to another. [2.6.8.2.2]

Meta IV

Meta IV is a formal notation for expressing the *abstract syntax* of a language. [1.5.1]

model

A *model* gives the mapping for *shorthand* notations expressed in terms of previously defined *concrete syntax*. [1.4.1, 1.4.2]

Modify!

Modify! is an implied *operator* on *structured sorts* for modifying a *field* from a *structured sort* and is implied in *expressions* when a *variable* is immediately followed by bracketed expressions and then $:=$. Within *axioms* *Modify!* is used explicitly. (See also *Extract!*). [5.3.1.10, 5.4.3.1, 5.4.3.2]

name

A *name* is a *lexical unit* used to name *SDL* objects. [2.2.1, 2.2.2]

natural

Natural is a *predefined data syntype* for which the *values* are the non-negative integers (i.e., 0, 1, 2, ...). The *operators* are the *operators* of the *sort integer*. [Annex D]

newtype

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]

nextstate

The *nextstate* terminates a transition and specifies the new *state* of the *process instance*. [2.8.6.2.1]

node

In the *abstract syntax*, a *node* is a designation of one of the basic concepts of *SDL*.

note

A *note* is text enclosed by */** and **/* which has no *SDL* defined semantics. See also *comment*. [2.2.1]

now expression

A *now expression* is an *expression* of the *time sort* which yields the current system time. [5.4.4.1]

null

Null is a *literal* of the *predefined data sort PId*. [Annex D]

offspring

offspring is an *expression* of the *predefined data sort PId*. When **offspring** is evaluated in a *process* it gives the *PId-value* 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]

operator

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, 5.2.1, 5.3.2]

operator signature

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

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

The *ordering operators* are <, <=, > or >=. [5.3.1.8]

out-connector

See *connector*.

outlet

An *outlet* represents a line, such as a *channel* or *flow line*, exiting a *macro diagram*. [4.2.2]

output

An *output* is an *action* within a *transition* which generates a *signal instance*. [2.7.4]

package

A *package* consists of a set of definitions. [2.4.1.2]

package interface

A *package interface* lists the visible definitions of a *package*. [2.4.1.2]

package reference

A *package reference* makes *definitions* within a *package* visible to another *package* or to a *system*. Either individual or all *definitions* listed in the *package interface* of the *package* become *visible*. [2.4.1.2]

page

A *page* is one of the components of a physical partitioning of a *diagram*. [2.2.5]

parameterized type

A *parameterized type* is a *type* where some of its dependencies on the enclosing *scope units* are represented by *context parameters*. [1.3.1]

parent

parent is an *expression* of the *predefined data sort PId*. 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]

partial type definition

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

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

PId is a *predefined data sort* 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*. [Annex D]

powerset

Powerset is the *predefined data generator* used to introduce mathematical sets. The *operators* for *powerset* are **in**, **Incl**, **Del**, **union**, **intersection** and the *ordering operators*. [Annex D]

predefined

Predefined is a *package* containing the *partial type definitions* for the *predefined data*. [2.4.1.2, 5.2.1, 5.3.1.3, Annex D]

predefined data

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*, *charstring*, *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 in the implicitly used *package Predefined*. [2.4.1.2, 5.1.1, 5.3.1.3, Annex D]

priority input

A *priority input* is a *shorthand notation* indicating that the reception of the *listed signals* should take precedence over *signals* listed in *inputs* to that *state*. [4.10]

procedure

A *procedure* is an encapsulation of the *behaviour* of part of a *process*. A *procedure* is defined in one place but may be referred to several times within a *process*. [2.4.6]

procedure call

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 constraint

A *procedure constraint* is the requirement on *actual parameters* to a formal *procedure context parameter* in terms of either a *procedure* or a *procedure constraint*. [6.2.2]

procedure context parameter

A *procedure context parameter* is a *context parameter* for which the *actual context parameter* must be a *procedure* fulfilling the *procedure constraint*. [6.2.2]

procedure signature

A *procedure signature* is a requirement on a formal *procedure context parameter* in terms of *constraints* on *formal parameters* of the *actual procedure*. [6.2.2]

procedure start

The *procedure start* in a *procedure* indicates the point at which execution of the *procedure* begins on a *procedure call*. [2.4.6]

process

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 constraint

A *process constraint* is the requirement on *actual parameters* to a formal *process context parameter*, either in terms of a *process type* (the *actual process instance* must then be a *process* of this *type* or of a *subtype*) or a *process signature* (the *actual process instance* must then be compatible with the *process signature*). [6.2.1]

process context parameter

A *process context parameter* is a *context parameter* for which the *actual parameter* must be a process definition or a typed process definition fulfilling the *process constraint*. [6.2.1]

process instance

A *process instance* is a member of the set of *processes*. A *process instance* is created at system creation time or dynamically as a result of a *create*. See **self**, **sender**, **parent**, and **offspring**. [2.4.4]

process signature

A *process signature* is a requirement on a formal *process context parameter* in terms of *constraints* on *formal parameters* of the actual processes and on the *valid input signal set* of the actual *process*. [6.2.1]

process type

A *process type* is the association of a *name* and a set of properties that all *process instances* of that *process type* will have. [6.1.1.3]

qualifier

The *qualifier* is part of an *identifier* which is the extra information to the *name* 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

Real is a *predefined data sort* 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 the *sort integer*. [Annex D]

referenced definition

A *referenced definition* is a syntactic means of distributing a *system definition* into several parts and relating the parts to each other. [2.4.1.3]

refinement

Refinement is the addition of new details to the functionality 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 procedure call

A *remote procedure call* is a request by a client *process* for a server *process* to execute a *procedure* defined in the server *process*. [4.14]

remote procedure input transition

A *remote procedure input transition* indicates a *state* in which a *process* will execute the specified *exported procedure*, possibly followed by a *transition*. [4.14]

remote procedure save

A *remote procedure save* indicates that the specified *exported procedure* will not be executed in the *state*. [4.14]

remote procedure definition

A *remote procedure definition* introduces the *name* and *procedure signature* for *imported* and *exported procedures*. [4.14]

remote variable definition

A *remote variable definition* introduces the *name* and *sort signature* for *imported* and *exported values*. [4.13]

reset

Reset is an operation defined for *timers* that makes the *timer* inactive and removes any pending *timer signals* from the *input port* of the *process*. See *active timer*. [2.8]

retained signal

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

A *return* (from a *procedure*) is the transfer of control to the calling *procedure* or *process*. [2.6.8.2.4]

revealed attribute

A *variable* owned by a *process* may have a *revealed attribute*, in which case another *process* is permitted to view the *value* associated with the *variable*. See *view definition*. [2.6.1.1]

save

A *save* is the declaration of those *signals* that should not be consumed in a given *state*. [2.6.5]

SDL

CCITT *SDL* (*Specification and Description Language*) is a formal language providing a set of constructs for the *specification* of the *behaviour* of a system.

SDL/GR

SDLIGR is the graphical representation in *SDL*. The *grammar for SDLIGR* is defined by the *concrete graphical grammar* and the *common textual grammar*. [1.2]

SDL/PR

SDLIPR is the textual phrase representation in *SDL*. The *grammar for SDL/PR* is defined by the *concrete textual grammar*. [1.2]

scope unit

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

Selection means providing those *external synonyms* needed to make a specific *system specification* from a generic *system specification*. [4.3, 4.3.3, 4.3.4]

self

self is an *expression* of the *predefined data sort* *PId*. 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

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

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

A *service* is an alternative way of specifying a part of the *behaviour* of a *process*. Each *service* may define a partial *behaviour* of a *process*. [2.4.5]

service type

A *service type* is the association of a *name* and a set of properties that all *services* of that *service type* will have. [6.1.1.4]

set

Set is an operation defined for *timers* which makes a *timer* an *active timer*. *Set* specifies the system time at which the *active timer* is to return a *timer signal*. [2.8]

shorthand notation

A *shorthand notation* is a *concrete syntax* notation providing a more compact representation implicitly referring to *Basic SDL* concepts. [1.4.2, 4.1]

signal

A *signal* is an instance of a *signal type* communicating information to a *process instance*. [2.5.4]

signal constraint

A *signal constraint* is the requirement on *actual parameters* to a formal *signal context parameter*, either in terms of a *signal type* (specifying the *signal definition* must be a *subtype* of the *constraint type*) or a *signal signature* (the actual *signal instance* must then be compatible with the *signal signature*). [6.2.4]

signal context parameter

A *signal context parameter* is a *context parameter* for which the *actual parameter* must be a *signal type* fulfilling the *signal constraint*. [6.2.4]

signal definition

A *signal definition* defines a *named signal type* and associates a list of zero or more *sort identifiers* with the *signal name*. This association allows *signals* to carry *values*. [2.5.4]

signal list

A *signal list* is a list of *signal identifiers* used in *channel* and *signal route definitions* to indicate which *signals* may be conveyed by the *channel* or *signal route* in one direction. [2.5.5]

signal route

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]

signal signature

A *signal signature* is a requirement on a formal *signal context parameter* in terms of *constraints* on *sorts* of the *parameters* of the actual *signal type*. [6.2.4]

signature

The *signature* of a *type* is the set of *sorts* and set of *operators* for that *type*. [5.2.1, 5.2.2]

simple expression

A *simple expression* is an *expression* which only contains *operators*, *synonyms*, and *literals* of the predefined *sorts*. [4.3.2]

sort

A *sort* is a set of *values* with common characteristics. *Sorts* are always nonempty and disjoint. [2.3.3, 5.1.2, 5.1.3, 5.2.1]

sort constraint

A *sort constraint* is the requirement on *actual parameters* to a formal *sort context parameter*, either in terms of a *sort* (the actual *sort* must then be a *subtype* of this *type*) or a *sort signature* (the actual *sort* must then be compatible with the operators in this *sort signature*). [6.2.9]

sort context parameter

A *sort context parameter* is a *context* for which the *actual parameter* must be a *sort* fulfilling the *sort constraint*. [6.2.9]

sort signature

A *sort signature* is a requirement on a formal *sort context parameter* in terms of constraints on *operators* and *literals* of the actual *sort*. [6.2.9]

specialization

Specialization creates a new type, called a *subtype*, by adding properties and/or redefining properties of another *type* called the *supertype*. [1.3.1]

specification

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]

spontaneous transition

A *spontaneous transition* allows interpretation of a *transition* without any *signal* consumption by the *process*. [2.6.6]

start

The *start* in a *process* or *service* is a *transition string* interpreted before any *state* or *action* at *process* creation. [2.6.2]

state

A *state* is a condition in which a *process instance* can consume a *signal*. [2.6.3]

stop

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. [2.6.8.2.3]

string

String is a *predefined data generator* used to introduce lists. The predefined *operators* include Length, First, Last, Substring and concatenation (//). [Annex D]

structure sort

A *structure sort* is a *sort* whose values are composed from a list of *values of sorts (fields)*. A *structure sort* has implicit *operators* and *equations* and special *concrete syntax* for these implicit *operators* by which the *values* of the *fields* can be accessed and modified independently. [5.3.1.10]

subblock

A *subblock* is a *block* contained within another *block*. *Subblocks* are formed when a *block* is *partitioned*. [3.2.1, 3.2.2]

subchannel

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

A *subsignal* is a *refinement* of a *signal* and may be further *refined*. [3.3]

subtype

A *subtype* is a *type* defined as a *specialization* of another *type* (the *supertype*). The properties associated with a *subtype* are all the properties of the *supertype*, except redefined *virtuals*, plus the added properties being special for the *subtype*. [1.3.1, 6.3]

supertype

A *supertype* is the *type* being used in defining a *subtype*. [1.3.1, 6.3]

symbol

A *symbol* is a terminal in the *concrete syntaxes*. A *symbol* may be one of a set of shapes in the *concrete graphical syntax*, or a sequence of characters in the *concrete textual syntax*. [1.5.2, 1.5.3]

synonym

A *synonym* is a *name* which represents a *value* of a *sort*. [5.3.1.13]

synonym context parameter

A *synonym context parameter* is a *context parameter* for which the *actual parameter* must be a *synonym* fulfilling the requirement on *actual parameters* to a formal *synonym context parameter* in terms of a *sort type*. [6.2.8]

syntype

A *syntype* specifies a set of *values* which corresponds to a subset of the *values* of the *sort*. The *operators* of the *syntype* are the same as those of the parent *sort*. [5.3.1.9]

system

A *system* is a set of *blocks* connected to each other and the *environment* by *channels*. [2.4.2]

system type

A *system type* is the association of a *name* and a set of properties that all *system instances* of this *type* will have. [6.1.1.1]

task

A *task* is an *action* within a *transition* containing either a sequence of *assignment statements* or *informal text*. [2.7.1]

term

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

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]

textual endpoint constraint

A *textual endpoint constraint* is the *SDL/PR* representation of an *endpoint constraint* of a *gate* putting requirements on which *blocks/processes/services* may be at the other end of a *channel/signal route/service signal route* connected to the *gate*. [6.1.4]

time

Time is a *predefined data sort* for which the *values* are denoted as the *values* of *real*. The predefined *operators* for *sorts time* and *duration* are + and −.

timer

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, 5.4.4.5]

timer active expression

A *timer active expression* is a *Boolean expression* which, when executed, indicates whether the identified *timer* is an *active timer*. [5.4.4.5]

timer context parameter

A *timer context parameter* is a *context parameter* for which the *actual parameter* must be a *timer* definition. [6.2.7]

transition

A *transition* is an active sequence which occurs when a *process instance* changes from one *state* to another. [2.6.8]

transition string

A *transition string* is a sequence of zero or more *actions*. [2.6.8.1]

type

A *type* is a set of properties for *instances*. Examples of kinds of *types* in *SDL* include *blocks*, *processes*, *services*, *signals*, and *systems*. [1.3.1]

type definition

A *type definition* defines the properties of a *type*. [1.3.1]

type expression

A *type expression* denotes a *type* that is either a base *type* or an anonymous *type* defined by applying *actual context parameter* to a parameterized base *type*. [6.1.2]

valid input signal set

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

A *valid specification* is a *specification* which follows the *concrete syntax* and static *well-formedness rules*. [1.3.3]

value

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]

value returning procedure

A *value returning procedure* is a *procedure* that may return a *value*. [5.4.5]

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 context parameter

A *variable context parameter* is a *context parameter* for which the *actual* must be a *variable* fulfilling the requirement on *actual parameters* to a formal *variable context parameter* in terms of a *sort type*; the *actual variable* must be a *variable* of this *type*. [6.2.5]

variable definition

A *variable definition* is the declaration that the *variable names* listed will be *visible* in the *process*, *procedure* or *service* containing the *variable definition*. [2.6.1.1]

view definition

A *view definition* defines a view of a variable 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

A *view expression* is used within an *expression* to yield the current *value* of a *viewed variable*. [5.4.4.4]

virtual continuous signal

A *virtual continuous signal* is a *continuous signal* where the *transition* may be redefined in a *subtype*. [4.11, 6.3.3]

virtual input

A *virtual input transition* is an *input transition* that in a *subtype* may be redefined to a *save* or to an *input transition*. [2.6.4, 6.3.3]

virtual priority input

A *virtual priority input* is a transition that may be redefined to a *priority input* or to a *save*. [6.3.3]

virtual procedure start

A *virtual procedure start* is a *transition* that may be redefined in a specialized *procedure*. [2.4.6, 6.3.3]

virtual remote procedure input

A *virtual remote procedure input* is a *transition* that may be redefined to a new *remote procedure input transition* or to a *remote procedure save*. [6.3.3]

virtual save

A *virtual save* is a *save* that in a *subtype* may be redefined to an *input transition*. [2.6.5, 6.3.3]

virtual start

A *virtual start* is a *start transition* that in a *subtype* may be redefined to a new *start transition*. [2.6.2, 6.3.3]

virtual spontaneous transition

A *virtual spontaneous transition* is a *spontaneous transition* in a *type* that may be redefined in a *subtype*. [2.6.6, 6.3.3]

virtual type

A *virtual type* is a *type* that may be redefined in *subtypes* of the enclosing *type*. [6.3.2]

virtuality

The *virtuality* of a *type* or *transition* indicates if the *type/transition* is a *virtual type*, redefined in a *subtype* (and still a *virtual type*), or finalized (that is redefined, but not a *virtual type*). [6.3.2]

virtuality constraint

The *virtuality constraint* of a *virtual type* puts requirements on the redefinitions of the *virtual type* by means of a *constraint type*. Both the definition and redefinitions of a *virtual type* must be definitions of *subtypes* of the *constraint type*. The *constraint type* of a *virtual type* also determines the properties of the *virtual type* known by the enclosing *type* having the local *virtual type*. [6.3.2]

visibility

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 kind* may have the same *name*. [2.2.2]

well-formedness rules

Well-formedness rules are constraints on an *abstract syntax* or *concrete syntax* enforcing static conditions not directly expressed by the syntax rules. [1.4.1, 1.4.2]