INTERNATIONAL TELECOMMUNICATION UNION

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Z.100
(11/99)

SERIES Z: LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

Formal description techniques (FDT) – Specification and Description Language (SDL)

Specification and description language (SDL)

ITU-T Recommendation Z.100

(Previously CCITT Recommendation)

ITU-T Z-SERIES RECOMMENDATIONS

LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

FORMAL DESCRIPTION TECHNIQUES (FDT)	
Specification and Description Language (SDL)	**Z.100–Z.109**
Application of Formal Description Techniques	Z.110–Z.119
Message Sequence Chart	Z.120–Z.129
PROGRAMMING LANGUAGES	
CHILL: The ITU-T high level language	Z.200–Z.209
MAN-MACHINE LANGUAGE	
General principles	Z.300–Z.309
Basic syntax and dialogue procedures	Z.310–Z.319
Extended MML for visual display terminals	Z.320–Z.329
Specification of the man-machine interface	Z.330–Z.399
QUALITY OF TELECOMMUNICATION SOFTWARE	Z.400–Z.499
METHODS FOR VALIDATION AND TESTING	Z.500–Z.599

For further details, please refer to ITU-T List of Recommendations.

ITU-T RECOMMENDATION Z.100

SPECIFICATION AND DESCRIPTION LANGUAGE (SDL)

Summary

Scope-objective

This Recommendation defines SDL (*Specification and Description Language*) intended for unambiguous specification and description of telecommunications systems. The scope of SDL is elaborated in clause 1. This Recommendation is a reference manual for the language.

Coverage

SDL has concepts for behaviour, data description and (particularly for larger systems) structuring. The basis of behaviour description is extended finite state machines communicating by messages. Data description is based on data types for values and objects. The basis for structuring is hierarchical decomposition and type hierarchies. These foundations of SDL are elaborated in the respective main clauses of this 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 1.2, but SDL is generally suitable for describing reactive systems. The range of application is from requirement description to implementation.

Status/Stability

This Recommendation is the complete language reference manual supported by guidelines for its usage in Supplement 1. Annex F gives a formal definition of SDL semantics. The main text of this Recommendation is stable and needs to be issued immediately to meet market needs, but further study is required to complete Annex F. Appendix I records the status of Z.100, and should be updated as further studies are completed. Although further language extensions are anticipated in the future, SDL-2000 as defined in this Recommendation should meet most user needs for some years. The current version is based on wide user experience of SDL and recent additional user needs.

The main text is accompanied by annexes:

–	Annex A	Index of non-terminals.
–	Annex B	Reserved for future use – Annex B (03/93) is no longer valid.
–	Annex C	Reserved for future use – Annex C (03/93) is no longer in force.
–	Annex D	SDL Predefined data.
–	Annex E	Reserved for examples.
–	Annex F	Formal Definition (further study needed for SDL-2000).
–	Appendix I	Status of Z.100, related documents and Recommendations.
–	Appendix II	Guidelines for the maintenance of SDL.
–	Appendix III	Systematic conversion of SDL-92 to SDL-2000.

ITU-T Z.100 has also an independently published supplement:

– Z.100 Supplement 1 SDL+ Methodology: Use of MSC and SDL (with ASN.1).

Associated work

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. For references to additional material on SDL, and information on industrial usage of SDL, see http://www.sdl-forum.org.

Background

Different versions of SDL have been recommended by CCITT and ITU-T since 1976. This version is a revision of Z.100 (03/93) and incorporates Addendum 1 to Z.100 (10/96) and parts of Z.105 (03/95).

Compared to SDL as defined in 1992, the version defined herein has been extended in the areas of object-oriented data, harmonization of a number of features to make the language simpler, and features to enhance the usability of SDL with other languages such as ASN.1, ITU-T ODL (Z.130), CORBA and UML. Other minor modifications have been included. Though care has been taken not to invalidate existing SDL documents; some changes may require some descriptions to be updated to use this version. Details on the changes introduced can be found in 1.5.

Source

ITU-T Recommendation Z.100 was revised by ITU-T Study Group 10 (1997-2000) and was approved under the WTSC Resolution No. 1 procedure on 19 November 1999.

Keywords

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

CONTENTS

Page

Page

Recommendation Z.100

SPECIFICATION AND DESCRIPTION LANGUAGE (SDL)

(revised in 1999)

1 Scope

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; that is its implementation.

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 used in this Recommendation for both required behaviour and actual behaviour.

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, tendering and design, while also allowing some issues to be left open;

c) may be extended to cover new developments;

d) is able to support several methodologies of system specification and design.

1.2 Application

This Recommendation is the reference manual for SDL. A methodology framework document, which gives examples of SDL usage, is available as Supplement 1 to Z.100 produced in the study period 1992-1996. Appendix I of Z.100 first published in March 1993 also contains methodology guidelines, though these do not exploit the full potential of SDL.

The main area of application for SDL is the specification of the behaviour of aspects of real time systems, and the design of such systems. Applications in the field of telecommunications include:

a) call and connection processing (for example, call handling, telephony signalling, metering) in switching systems;

b) maintenance and fault treatment (for example alarms, automatic fault clearance, routine tests) in general telecommunications systems;

c) system control (for example, overload control, modification and extension procedures);

d) operation and 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; that is where 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) ITU-T Recommendations, or other similar Standards (international, regional or national);

d) system design specifications;

e) detailed specifications;

f) system design descriptions (both high level and detailed enough to directly produce implementations);

g) system testing descriptions (in particular in combination with MSC and TTCN).

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

1.3 System specification

An SDL specification defines 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 that 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.

1.4 Differences between SDL-88 and SDL-92

The language defined in the previous version of this Recommendation was an extension of Z.100 as published in the 1988 *Blue Book*. The language defined in the *Blue Book* is known as SDL-88 and the language defined in the previous version of this Recommendation was called SDL-92. Every effort had 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 ITU-T member-bodies.

The major extensions were in the area of object orientation. While SDL-88 is object based in its underlying model, some language constructs had been added to allow SDL-92 to more completely and uniformly support the object paradigm:

a) packages;

b) system, block, process and service types;

c) system, block, process and service (set of) instances based on types;

d) parameterization of types by means of context parameters;

e) specialization of types, and redefinition of virtual types and transitions.

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

In a few cases, changes were made to SDL-88 where the definition of SDL-88 was not consistent. The restrictions and changes introduced can be overcome by an automatic translation procedure. This procedure was also necessary, to convert an SDL-88 document in SDL-92 that contained names consisting of words which are keywords of SDL-92.

For the **output** construct the semantics were simplified between SDL-88 and SDL-92, and this may have invalidated 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 were changed.

For the **export**/**import** construct an optional remote variable definition was introduced, in order to align export of variables with the introduced export of procedures (remote procedure). This necessitated a change to any SDL-88 document, which contained qualifiers in import expressions or introduced several imported names in the same scope with different sorts. In the (rare) cases where it was necessary to qualify import variables to resolve resolution by context, the change to make SDL-88 into SDL-92 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 had been made local to the viewing process or service. This necessitated a change to SDL-88 documents, which contained qualifiers in view definitions or in view expressions. To make SDL-88 into SDL-92 is to remove these qualifiers. This did not change the semantics of the view expressions, since these are decided by their (unchanged) pid expressions.

The **service** construct was defined as a primitive concept, instead of being a shorthand, without extending its properties. The use of service was 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 was deleted, signal routes could be used instead. This was only a minor conceptual change, and had no implications for concrete use (the syntax of SDL-88 service signal route and SDL-92 signal route were 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 were extended considerably, e.g. **signal** definition. It should be noted that the extensions were optional, but were used for utilising the power introduced by the object oriented extensions, e.g. to use parameterization and specialization for signals.

Keywords of SDL-92 that 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.

1.5 Differences between SDL-92 and SDL-2000

A strategic decision was made to keep SDL stable for the period 1992 to 1996, so that at the end of this period only a limited number of changes were made to SDL. These were published as Addendum 1 to Z.100 (10/96) rather than updating the SDL-92 document. Although this version of

SDL was sometimes called SDL-96, it was small change compared with the change from SDL-88 to SDL-92. The changes were:

a) harmonizing signals with remote procedures and remote variables;

b) harmonizing channels and signal routes;

c) adding external procedures and operations;

d) allowing a block or process to be used as a system;

e) state expressions;

f) allowing packages on blocks and processes;

g) parameterless operators.

These have now been incorporated into Z.100, together with a number of other changes to produce a version of SDL known as SDL-2000. In this Recommendation the language defined by Z.100 (03/93) with Addendum 1 to Z.100 (10/96) is still called SDL-92.

The advantages of language stability, which was maintained over the period from 1992 to 1996, began to be outweighed by the need to update SDL to support and better match other languages that are frequently used in combination with SDL. Also modern tools and techniques have made it practical to generate software more directly from SDL specifications, but further significant gains could be made by incorporating better support for this use in SDL. While SDL-2000 is largely an upgrade of SDL-92, it was agreed that some incompatibility with SDL-92 was justified; otherwise the resulting language would have been too large, too complex and too inconsistent. This subclause provides information about the changes. How most SDL-92 descriptions might be systematically transformed into SDL-2000 is given in Appendix III.

Changes have been made in a number of areas, which focus on simplification of the language, and adjustment to new application areas:

a) adjustment of syntactical conventions to other languages with which SDL is used;

b) harmonization of the concepts of system, block and process to be based on "agent" , and merging of the concept of signal route into the concept channel;

c) interface descriptions;

d) exception handling;

e) support for textual notation of algorithms within SDL/GR;

f) composite states;

g) replacement of the service construct with the state aggregation construct;

h) new model for data;

i) constructs to support the use of ASN.1 with SDL previously in Z.105 (03/95).

Other changes are: nested packages, direct containment of blocks and processes in blocks, **out**-only parameters.

On the syntactic level, SDL-2000 is case-sensitive. Keywords are available in two spellings: all uppercase or all lowercase. Keywords of SDL-2000 that are not keywords of SDL-92 are:

abstract, aggregation, association, break, choice, composition, continue, endexceptionhandler, endmethod, endobject, endvalue, exception, exceptionhandler, handle, method, object, onexception, ordered, private, protected, public, raise, value.

The following keywords of SDL-92 are not keywords in SDL-2000:

all, axioms, constant, endgenerator, endnewtype, endrefinement, endservice, error, fpar, generator, imported, literal, map, newtype, noequal, ordering, refinement, returns, reveal, reverse, service, signalroute, view, viewed.

A small number of constructs of SDL-92 are not available in SDL-2000: view expression, generators, block substructures, channel substructures, signal refinement, axiomatic definition of data, macro diagrams. These constructs were rarely (if ever) used, and the overhead of keeping them in the language and tools did not justify their retention.

2 References

The following ITU-T Recommendations and other references contain provisions which, through reference in this text, constitute provisions of this Recommendation. At the time of publication, the editions indicated were valid. All Recommendations and other references are subject to revision; all users of this Recommendation are therefore encouraged to investigate the possibility of applying the most recent edition of the Recommendations and other references listed below. A list of the currently valid ITU-T Recommendations is regularly published.

– CCITT Recommendation T.50 (1992), *International Reference Alphabet (IRA) (Formerly International Alphabet No. 5 or IA5) – Information technology – 7-bit coded character set for information interchange.*

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

3 Definitions

The are numerous terms defined throughout this Recommendation and a list of definitions in this clause would be a repetition of much of the text of the Recommendation. Therefore only a few key terms are given in this clause.

3.1 agent: The term agent is used to denote a system, block or process that contains one or more extended finite state machines.

3.2 block: A block is an agent that contains one or more concurrent blocks or processes and may also contain an extended finite state machine that owns and handles data within the block.

3.3 body: A body is a state machine graph of an agent, procedure, composite state, or operation.

3.4 channel: A channel is a communication path between agents.

3.5 environment: The environment of the system is everything in the surroundings that communicates with the system in an SDL-like way.

3.6 gate: A gate represents a connection point for communication with an agent type, and when the type is instantiated it determines the connection of the agent instance with other instances.

3.7 instance: An instance is an object created when a type is instantiated.

3.8 object: The term object is used for data items that are references to values.

3.9 pid: The term pid is used for data items that are references to agents.

3.10 procedure: A procedure is an encapsulation of part of the behaviour of an agent, that is defined in one place but may be called from several places within the agent. Other agents can call a remote procedure.

3.11 process: A process is an agent that contains an extended finite state machine, and may contain other processes.

3.12 signal: The primary means of communication is by signals that are output by the sending agent and input by the receiving agent.

3.13 sort: A sort is a set of data items that have common properties.

3.14 state: An extended finite state machine of an agent is in a state if it is waiting for a stimulus.

3.15 stimulus: A stimulus is an event that can cause an agent that is in a state to enter a transition.

3.16 system: A system is the outermost agent that communicates with the environment.

3.17 timer: A timer is an object owned by an agent that causes a timer signal stimulus to occur at a specified time.

3.18 transition: A transition is a sequence of actions an agent performs until it enters a state.

3.19 type: A type is a definition that can be used for the creation of instances, and can also be inherited and specialized to form other types. A parameterized type is a type that has parameters. When these parameters are given different actual parameters, different unparameterized types are defined that when instantiated give instance with different properties.

3.20 value: The term value is used for the class of data that is accessed directly. Values can be freely passed between agents.

4 Abbreviations

This Recommendation uses the following abbreviations:

SDL/GR	Graphic Representation form of SDL
SDL/PR	Textual Phrase Representation form of SDL
SDL-2000	SDL as defined by this Recommendation
SDL-92	SDL as defined by Z.100 (03/93) with Addendum 1 (10/96)
SDL-88	SDL as defined by Z.100 (1988).

5 Conventions

The text of this clause is not normative. Instead it defines the conventions used for describing SDL. The usage of SDL in this clause is only illustrative. The metalanguages and conventions introduced are solely introduced for the purpose of describing SDL unambigously.

5.1 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.

Although SDL can be written in either SDL/PR or SDL/GR, the language has been designed with the knowledge that SDL/PR is used infrequently for purposes such as interchanging between tools. The common interchange format specified in Z.106 (10/96) further diminishes the use of SDL/PR. Most users use SDL/GR.

Figure 5-1 shows the relationships between SDL/PR, SDL/GR, the concrete grammars and the abstract grammar.

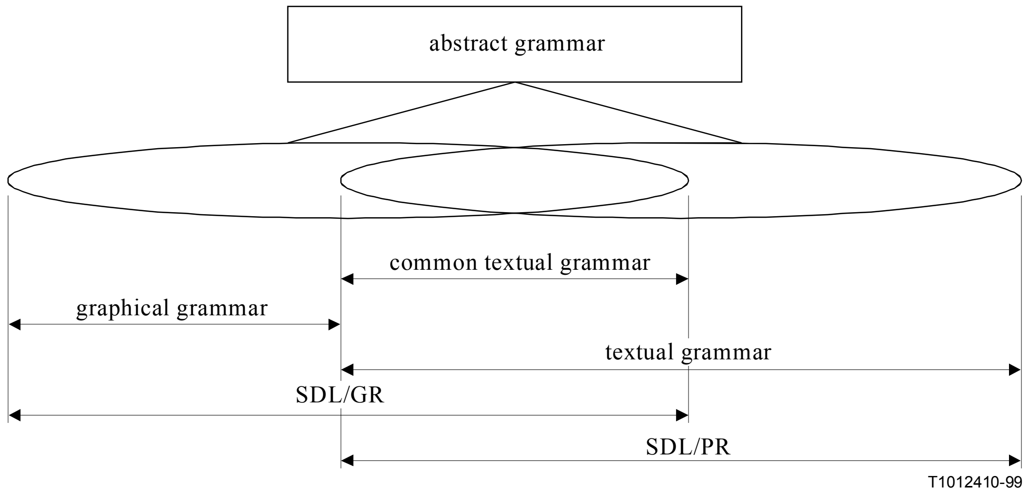


Figure 5-1/Z.100 SDL Grammars

Each of the concrete grammars has a definition of its own syntax and of its relationship to the abstract grammar (that is, 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 inherits the semantics via its relationship 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.

For some constructs there is no directly equivalent abstract syntax. In these cases, a model is given for the transformation from concrete syntax into the concrete syntax of other constructs that (directly or indirectly via further models) have an abstract syntax. Items that have no mapping to the abstract syntax (such as comments) do not have any formal meaning.

5.2 Basic definitions

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

5.2.1 Definition, type and instance

In this Recommendation, the concepts of type and instance and their relationship are fundamental. The schema and terminology defined below and shown in Figure 5-2 are used.

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

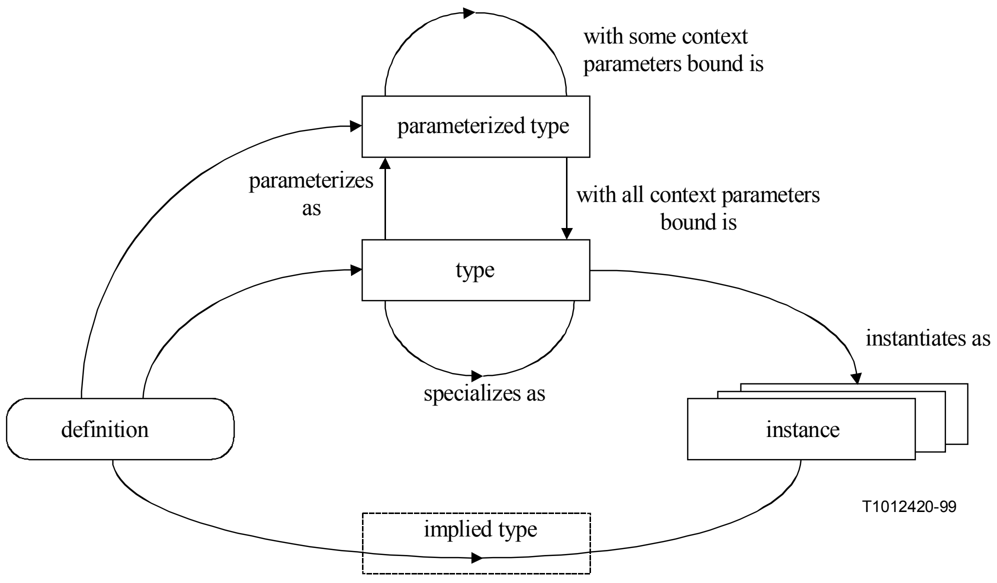


Figure 5-2/Z.100 – The type concept

Definitions introduce named entities, which are either types or instances with implied types. A definition of a type defines all properties associated with that type. An example of an instance definition is a process definition. An example of a definition that 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 is 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 type derived from it.

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...." .

5.2.2 Environment

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

It is assumed that there are one or more agent instances in the environment, and therefore stimuli flowing from the environment towards the system have associated identities of these agent instances. These agents have pids that are distinguishable from any of the pid within the system (see 12.1.6).

Although the behaviour of the environment is non-deterministic, it is assumed to obey the constraints given by the system specification.

5.2.3 Error

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. Predefined exceptions (see D.3.16) will be raised, when an error is encountered during the interpretation of a system. If the exception is not handled, the subsequent behaviour of the system cannot be derived from the specification.

5.3 Presentation style

The following presentation style is used to separate the different language issues under each topic.

5.3.1 Division of text

This 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 6.5).

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 notations that do not have a direct abstract syntax and modelled in terms of other concrete syntax constructs. A notation that is modelled by other constructs is known as a shorthand, and is considered to be derived syntax for the transformed form.

f) *Examples*

In a few cases (such as <sdl specification>) a titled enumeration item, *Concrete grammar*, is used where textual and graphical syntax are merged, or where much of the grammar rules would have to be repeated for textual and graphical cases. The mechanism for merging SDL/PR and SDL/GR in these cases is not defined by this Recommendation.

5.3.2 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 this Recommendation presented only to aid understanding and not to make this Recommendation complete.

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

The remainder of this subclause 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 5.4.1.

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 clause.

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 that 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 5.4.2.

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

In many cases there is a simple relationship between the concrete and abstract syntax, because the 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 (see 5.4.2) are significant.

Concrete textual syntax that is not a shorthand form 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 that 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 5.4.3.

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 that 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 that 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 (7.3), and macros (6.2), must be considered before the correspondence between the concrete and the abstract syntax. These expansions are detailed in Annex F.

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-identifier*s (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 6.3. 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 processes and/or blocks.

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, this Recommendation describes how the instances are created from the system definition and how these instances are interpreted within an "abstract SDL machine". Lists are denoted in the Abstract Syntax by the suffixes "*" and "+" (see 5.4.1 and 5.4.2).

Dynamic conditions are conditions that must be satisfied during interpretation and cannot be checked without interpretation. Dynamic conditions may lead to errors (see 5.2.3).

NOTE – Behaviour of the system is produced by "interpreting" the SDL. The word "interpret" is explicitly chosen (rather than an alternative such as "executed") to include both mental interpretation by a human and the interpretation of the SDL by a computer.

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.

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 Annex F. The transformation order is also followed when defining the concepts in this clause.

The result of the transformation of a fragment of text in derived concrete syntax is usually either another fragment of text in derived concrete syntax, or a fragment of text in concrete syntax. The result of the transformation may also be empty. In the latter case, the original text is removed from the specification. When discussing the model for a concrete syntax, the meta-symbol ***transform*** may be used and refers to the result of the transformation of its argument. For example,

<expression>-***transform***

refers to the transformation of <expression>.

Examples

If there is a titled enumeration item *Example(s)* which contains example(s). More examples will be provided in Annex E. Examples are informative.

5.4 Metalanguages

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

The grammar given in this Recommendation has been written to aid the presentation in this Recommendation so that the rule names are meaningful in the context they are given and can be used in text. This means that there are a number of apparent ambiguities that can easily be resolved by systematic rewriting of the syntax rules or the application of semantic rules.

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

5.4.1 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 channel definition is:

Channel-path	::	*Originating-gate*
		Destination-gate
		*Signal-identifier**-set***

which defines the domain for the composite object (tree) named *Channel-path.* This object consists of three sub-components, which in turn might be trees.

The Meta IV definition

Agent-identifier	=	*Identifier*

expresses that an *Agent-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) Natural objects

   Example:

Number-of-instance	::	*Nat* [*Nat*]

   Number-of-instances denotes a composite domain containing one mandatory natural (*Nat*) value and one optional natural ([*Nat*]) 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:

Channel-definition	::	*Channel-name*
		[**NODELAY**]
		*Channel-path**-set***

   A channel may not be delaying. This is denoted by an optional quotation **NODELAY**.

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:

   Name :: *Token*

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

d) Unspecified objects

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

   Example:

   Informal-text :: ...

   Informal-text contains an object that is not interpreted.

The following operators (constructors) in BNF (see 5.4.2) have the same use in the Meta IV abstract syntax: "*" for a possibly empty list, "+" for a non-empty list, "|" for an alternative, and "[" "]" for optionality.

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

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

Example

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

An *Agent-graph* consists of an *Agent-start-node* and a set of *State-nodes*.

5.4.2 BNF

In the Backus-Naur Form (BNF) for lexical rules, the terminals are <space> and the printed characters in 6.1.

In the Backus-Naur Form for non-lexical rules, a terminal symbol is one of the lexical units defined in 6.1 (<name>, <quoted operation name>, <character string>, <hex string>, <bit string>, <special>, <composite special> or <keyword>). In non-lexical rules, a terminal can be represented by one of the following:

a) a keyword (such as state);

b) the character for the lexical unit if it consists of a single character (such as "=");

c) the lexical unit name (such as <quoted operation name> or <bit string>);

d) the name of a <composite special> lexical unit (such as <implies sign>).

To avoid confusion with BNF grammar, the lexical unit names <asterisk>, <plus sign>, <vertical line>, <left square bracket>, <right square bracket>, <left curly bracket> and <right curly bracket> are always used rather than the equivalent characters. 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 lexical units. 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,

```
<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, <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. For example, <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,

```
<diagram in package> ::=
                         |     <package diagram>
                         |     <package reference area>
                         |     <type in agent area>
                         |     <data type reference area>
                         |     <signal reference area>
                         |     <procedure reference area>
                         |     <interface reference area>
                         |     <create line area>
                         |     <option area>
```

expresses that a <diagram in package> is a <package diagram>, or a <package reference area>, or a <type in agent area>, or a <data type reference area>, or a <signal reference area>, or a <procedure reference area>, or an <interface reference area>, or a <create line area> or an <option area>.

Syntactic elements may be grouped together by using curly brackets ("{" and "}"), similar to the parentheses in Meta IV (see 5.4.1). A curly bracketed group may contain one or more vertical bars, indicating alternative syntactic elements. For example,

```
<operation definitions> ::=
                              {     <operation definition>
                              |     <textual operation reference>
                              |     <external operation definition> }+
```

Repetition of syntactic elements or 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 <operation definitions> contains at least one <operation definition> or <textual operation reference> or <external operation definition>, and may contain more than one of any of these.

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

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

expresses that a <valid input signal set> may, but need not, contain <signal list>.

5.4.3 Metalanguage for graphical grammar

For the graphical grammar the metalanguage described in 5.4.2 is extended with the following metasymbols:

a) ***contains***

b) ***is associated with***

c) ***is followed by***

d) ***is connected to***

e) ***set***

The ***set*** metasymbol is a postfix operator operating on the immediately preceding syntactic elements within curly brackets, and indicating an (unordered) set of items. Each item may be any group of syntactic elements, in which case it must be expanded before applying the ***set*** metasymbol.

Example:

{ <operation text area>* <operation graph area> } ***set***

is a set consisting of zero or more <operation text area>s, and one <operation graph area>. The ***set*** metasymbol is used when the position of the syntactic elements relative to one another in the diagram is irrelevant and the elements can be considered in any order.

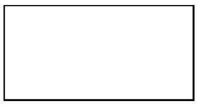
All the other metasymbols are infix operators, having a graphical non-terminal symbol as the left-hand argument. The right-hand argument is either a group of syntactic elements within curly brackets or a single syntactic element. If the right-hand side of a production rule has a graphical non-terminal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

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

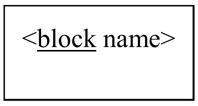
<block reference area> ::=

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

<block symbol> ::=



means the following



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

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

The metasymbol ***is connected to*** means that its right-hand argument is connected (both logically and in drawing) to its left-hand argument and implies a flow line symbol (see 6.5).

6 General rules

6.1 Lexical rules

Lexical rules define lexical units. Lexical units are the terminal symbols of the *Concrete textual grammar* and *Concrete graphical grammar*.

```
<lexical unit> ::=
                          <name>
                  |       <quoted operation name>
                  |       <character string>
                  |       <hex string>
                  |       <bit string>
                  |       <note>
                  |       <composite special>
                  |       <special>
                  |       <keyword>

<name> ::=
                          <underline>* <word> {<underline>+ <word>}* <underline>*
                  |       {<decimal digit>}+ { {<full stop>} <decimal digit>+ }*

<word> ::=
                          {<alphanumeric>}+

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

<letter> ::=
                          <uppercase letter> | <lowercase letter>

<uppercase 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

<lowercase 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

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

<quoted operation name> ::=
                          <quotation mark> <infix operation name> <quotation mark>
                  |       <quotation mark> <monadic operation name> <quotation mark>

<infix operation name> ::=
                          or    |    xor    |    and    |    in    |    mod    |    rem
                  |       <plus sign>                 |    <hyphen>
                  |       <asterisk>                  |    <solidus>
                  |       <equals sign>               |    <not equals sign>
                  |       <greater than sign>         |    <less than sign>
                  |       <less than or equals sign>  |    <greater than or equals sign>
                  |       <concatenation sign>        |    <implies sign>

<monadic operation name> ::=
                          <hyphen>    |    not

<character string> ::=
                          <apostrophe> { <general text character>
                                       | <special>
                                       | <apostrophe> <apostrophe>
                                       }* <apostrophe>
```

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

```
<hex string> ::=
                        <apostrophe> { <decimal digit>
                                     | a   | b   | c   | d   | e   | f
                                     | A   | B   | C   | D   | E   | F
                                     }* <apostrophe>  { H | h }

<bit string> ::=
                        <apostrophe> { 0   | 1
                                     }* <apostrophe> { B | b }

<note> ::=
                        <solidus> <asterisk><note text> <asterisk>+ <solidus>

<note text> ::=
                        {     <general text character>
                        |     <other special>
                        |     <asterisk>+ <not asterisk or solidus>
                        |     <solidus>
                        |     <apostrophe> }*

<not asterisk or solidus> ::=
                        <general text character> | <other special> | <apostrophe>

<text> ::=
                        { <general text character> | <special> | <apostrophe> }*

<general text character> ::=
                        <alphanumeric> | <other character> | <space>

<composite special> ::=
                   |    <result sign>
                   |    <composite begin sign>
                   |    <composite end sign>
                   |    <concatenation sign>
                   |    <history dash sign>
                   |    <greater than or equals sign>
                   |    <implies sign>
                   |    <is assigned sign>
                   |    <less than or equals sign>
                   |    <not equals sign>
                   |    <qualifier begin sign>
                   |    <qualifier end sign>

<result sign> ::=
                        <hyphen> <greater than sign>

<composite begin sign> ::=
                        <left parenthesis> <full stop>

<composite end sign> ::=
                        <full stop> <right parenthesis>

<concatenation sign> ::=
                        <solidus> <solidus>

<history dash sign> ::=
                        <hyphen> <asterisk>

<greater than or equals sign> ::=
                        <greater than sign> <equals sign>

<implies sign> ::=
                        <equals sign><greater than sign>

<is assigned sign> ::=
                        <colon> <equals sign>

<less than or equals sign> ::=
                        <less than sign> <equals sign>
```

```
<not equals sign> ::=
                        <solidus> <equals sign>

<qualifier begin sign> ::=
                        <less than sign> <less than sign>

<qualifier end sign> ::=
                        <greater than sign> <greater than sign>

<special> ::=
                        <solidus>     |     <asterisk>     |     <other special>

<other special> ::=
                        <exclamation mark>     |     <number sign>
                  |     <left parenthesis>     |     <right parenthesis>
                  |     <plus sign>            |     <comma>               |     <hyphen>
                  |     <full stop>            |     <colon>               |     <semicolon>
                  |     <less than sign>       |     <equals sign>         |     <greater than sign>
                  |     <left square bracket>  |     <right square bracket>
                  |     <left curly bracket>   |     <right curly bracket>

<other character> ::=
                        <quotation mark>   |     <dollar sign>        |     <percent sign>
                  |     <ampersand>        |     <question mark>      |     <commercial at>
                  |     <reverse solidus>  |     <circumflex accent> |     <underline>
                  |     <grave accent>     |     <vertical line>      |     <tilde>

<exclamation mark>      ::=   !

<quotation mark>        ::=   "

<left parenthesis>      ::=   (

<right parenthesis>     ::=   )

<asterisk>              ::=   *

<plus sign>             ::=   +

<comma>                 ::=   ,

<hyphen>                ::=   -

<full stop>             ::=   .

<solidus>               ::=   /

<colon>                 ::=   :

<semicolon>             ::=   ;

<less than sign>        ::=   <

<equals sign>           ::=   =

<greater than sign>     ::=   >

<left square bracket>   ::=   [

<right square bracket>  ::=   ]

<left curly bracket>    ::=   {

<right curly bracket>   ::=   }

<number sign>           ::=   #

<dollar sign>           ::=   $

<percent sign>          ::=   %

<ampersand>             ::=   &

<apostrophe>            ::=   '

<question mark>         ::=   ?

<commercial at>         ::=   @
```

```
<reverse solidus>        ::=   \

<circumflex accent>      ::=   ^

<underline>              ::=   _

<grave accent>           ::=   `

<vertical line>          ::=   |

<tilde>                  ::=   ~

<keyword>::=
          abstract                 |  active                 |  adding
       |  aggregation              |  alternative            |  and
       |  any                      |  as                     |  association
       |  atleast                  |  block                  |  break
       |  call                     |  channel                |  choice
       |  comment                  |  composition            |  connect
       |  connection               |  constants              |  continue
       |  create                   |  dcl                    |  decision
       |  default                  |  else                   |  endalternative
       |  endblock                 |  endchannel             |  endconnection
       |  enddecision              |  endexceptionhandler    |  endinterface
       |  endmacro                 |  endmethod              |  endobject
       |  endoperator              |  endpackage             |  endprocedure
       |  endprocess               |  endselect              |  endstate
       |  endsubstructure          |  endsyntype             |  endsystem
       |  endtype                  |  endvalue               |  env
       |  exception                |  exceptionhandler       |  export
       |  exported                 |  external               |  fi
       |  finalized                |  for                    |  from
       |  gate                     |  handle                 |  if
       |  import                   |  in                     |  inherits
       |  input                    |  interface              |  join
       |  literals                 |  macro                  |  macrodefinition
       |  macroid                  |  method                 |  methods
       |  mod                      |  nameclass              |  nextstate
       |  nodelay                  |  none                   |  not
       |  now                      |  object                 |  offspring
       |  onexception              |  operator               |  operators
       |  optional                 |  or                     |  ordered
       |  out                      |  output                 |  package
       |  parent                   |  priority               |  private
       |  procedure                |  protected              |  process
       |  provided                 |  public                 |  raise
       |  redefined                |  referenced             |  rem
       |  remote                   |  reset                  |  return
       |  save                     |  select                 |  self
       |  sender                   |  set
       |  signal                   |  signallist             |  signalset
       |  size                     |  spelling               |  start
       |  state                    |  stop                   |  struct
       |  substructure             |  synonym                |  syntype
       |  system                   |  task                   |  then
       |  this                     |  timer                  |  to
       |  try                      |  type                   |  use
       |  value                    |  via                    |  virtual
       |  with                     |  xor

<space> ::=
```

The characters in <lexical unit>s and in <note>s as well as the character <space> and control characters are defined by the International Reference Version of the International Reference Alphabet (Recommendation T.50). The lexical unit <space> represents the T.50 SPACE character (acronym SP), which (for obvious reasons) cannot be shown.

<text> is used in a <comment area> where it is equivalent to a <character string> and in a <text extension area> where it must be treated as a sequence of other lexical units.

T.50 delete characters are completely ignored. If an extended character set is used, the characters that are not defined by T.50 may only appear in <text> in a <comment area> or a <character string> in a <comment> or within a <note>.

When an <underline> character is followed by one or more <space>s or control characters, 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. This rule is applied before any other lexical rule.

A (non-space) control character may appear where a <space> may appear, and has the same meaning as a <space>.

An occurrence of a control character is not significant in <informal text> and in <note>. In order to construct a string expression containing control characters, the <concatenation sign> operator and the literals for control characters must be used. All spaces in a character string are significant: a sequence of spaces is not treated as one space.

Any number of <space>s may be inserted before or after any <lexical unit>. Inserted <space>s or <note>s have no syntactic relevance, but sometimes a <space> or <note> is needed to separate one <lexical unit> from another.

In all <lexical unit>s except keywords, uppercase <letter>s and lowercase letters are distinct. Therefore AB, aB, Ab and ab represent four different <word>s. An all uppercase <keyword> has the same use as the all lowercase <keyword> with the same spelling (ignoring case), but a mixed case letter sequence with the same spelling as a <keyword> represents a <word>.

For conciseness within the Lexical Rules, *Concrete textual grammar* and *Concrete graphical grammar*, the lowercase <keyword> as a terminal denotes that the uppercase <keyword> with the same spelling and may be used at the same place. For example, the concrete syntax terminator

endblock

represents the lexical alternatives

{ **endblock** | **ENDBLOCK** }

NOTE – Boldface lower case is used for keywords within this Recommendation. Distinguishing by font attributes is not a mandatory requirement, but can be useful to the readers of a specification.

A <lexical unit> is terminated by the first character, which cannot be part of <lexical unit> according to the syntax specified above. If a <lexical unit> can be both a <name> and a <keyword>, then it is a <keyword>. If two <quoted operation name>s differ only in case, the semantics of the lowercase name applies, so that (for example) the expression "OR"(a, b) means the same as (a **or** b).

Special lexical rules apply within a <macro body>.

6.2 Macro

A macro definition contains a collection of lexical units, which can be included in one or more places in the concrete textual grammar of an <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.

6.2.1 Additional Lexical rules

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

6.2.2 Macro definition

```
<macro definition> ::=
                    macrodefinition <macro name>
                         [<macro formal parameters>] <end>
                         <macro body>
                    endmacro [<macroname>] <end>

<macro formal parameters> ::=
                    ( <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**.

A <macro definition> contains lexical units.

<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 an example of a <macro definition>:

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

6.2.3 Macro call

<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>, the <macro actual parameter> must be a <character string>. If the <macro actual parameter> is a <character string>, the result 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.

Model

An <sdl specification> may contain macro definitions and macro calls. Before such an <sdl specification> 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 is expanded to replace the macro call. This means that a copy of the macro body 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>.

Example

Below is 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 6.2.2, gives the following result.

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

6.3 Visibility rules, names and identifiers

Abstract grammar

Identifier	::	*Qualifier Name*
Qualifier	=	*Path-item* +
Path-item	=	*Package-qualifier*
	|	*Agent-type-qualifier*
	|	*Agent-qualifier*
	|	*State-type-qualifier*
	|	*State-qualifier*
	|	*Data-type-qualifier*
	|	*Procedure-qualifier*
	|	*Signal-qualifier*
	|	*Interface-qualifier*
Package-qualifier	::	*Package-name*
Agent-type-qualifier	::	*Agent-type-name*
Agent-qualifier	::	*Agent-name*

State-type-qualifier	::	*State-type-name*
State-qualifier	::	*State-name*
Data-type-qualifier	::	*Data-type-name*
Procedure-qualifier	::	*Procedure-name*
Signal-qualifier	::	*Signal-name*
Interface-qualifier	::	*Interface-name*
Package-name	=	*Name*
Agent-type-name	=	*Name*
Agent-name	=	*Name*
State-type-name	=	*Name*
Data-type-name	=	*Name*
Interface-name	=	*Name*
Name	::	*Token*

Concrete textual grammar

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

<qualifier> ::=
                    <qualifier begin sign> <path item> { / <path item> }* <qualifier end sign>

<string name> ::=
                    <character string>
               |    <bit string>
               |    <hex string>

<path item> ::=
                    [<scope unit kind>] <name>

<scope unit kind> ::=
                    package
               |    system type
               |    system
               |    block
               |    block type
               |    process
               |    process type
               |    state
               |    state type
               |    procedure
               |    signal
               |    type
               |    operator
               |    method
               |    interface
```

Scope units are defined by the following non-terminal symbols of the concrete grammar. Symbols belonging to the concrete textual grammar are shown in the left-hand column; the corresponding symbols of the concrete graphical grammar are shown in the right-hand column.

<package definition>	<package diagram>
<agent definition>	<agent diagram>
<agent type definition>	<agent type diagram>
<procedure definition>	<procedure diagram>
<data type definition>	
<interface definition>	
<operation definition>	<operation diagram>
<composite state>	<composite state area>
<composite state type definition>	<composite state type diagram>
<sort context parameter>	
<signal definition>	
<signal context parameter>	
<compound statement>	<task area>

A scope unit has a list of definitions attached. Each of the definitions defines one or more entities belonging to a certain entity kind and having an associated name, including <textual gate definition>s, <formal context parameter>s, <agent formal parameters>s, and <formal variable parameters> contained in the scope unit.

Although <quoted operation name>s and <string name>s have their own syntactical notation, they are in fact <name>s that represent *Name*s in the *Abstract syntax*. In the following, they are treated as if they were syntactically also <name>s.

Entities can be grouped into entity kinds. The following entity kinds exist:

a) packages;

b) agents (system, blocks, processes);

c) agent types (system types, block types, process types);

d) channels, gates;

e) signals, timers, interfaces, data types;

f) procedures, remote procedures;

g) variables (including formal parameters), synonyms;

h) literals, operators, methods;

i) remote variables;

j) sorts;

k) state types;

l) signal lists;

m) exceptions.

A formal context parameter is an entity of the same entity kind as the corresponding actual context parameters.

A reference definition is an entity after the transformation step for <referenced definition> (see Annex F).

Each entity is said to have its defining context in the scope unit that defines it.

Entities are referenced by means of <identifier>s. The <qualifier> within an <identifier> specifies uniquely the defining context of the entity.

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. The *Name* of an entity is then represented by the qualifier, the name of the entity, and, only for entities of kind h), the signature (see 12.1.7.1, 12.1.4). All entities of the same kind must have different *Name*s.

NOTE 1 – Consequently, no two definitions in the same scope unit and belonging to the same entity kind can have the same <name>. The only exceptions are operations defined in the same <data type definition>, as long as they differ in at least one argument <sort> or the result <sort>.

<<u>state</u> name>s, <<u>connector</u> name>s, and <<u>gate</u> name>s occurring in channel definitions, <macro formal parameter>s and <<u>macro</u> name>s have special visibility rules and cannot be qualified. Other special visibility rules are explained in the appropriate clauses.

NOTE 2 – There is no <scope unit kind> corresponding to the scope units defined by the <task>, <task area>, and <compound statement> schemata. Therefore, it is not possible to refer to the identifiers introduced in a definition attached to these scope units by qualifiers.

An entity can be referenced by using an <identifier>, if the entity is visible. An entity is visible in a scope unit if:

a) it has its defining context in that scope unit; or

b) the scope unit is a specialization and the entity is visible in the base type; and
   1) it is not protected from visibility by a special construction defined in 12.1.9.3; and
   2) data specialization renaming has not been applied (12.1.3); and
   3) it is not a formal context parameter which has already been bound to an actual context parameter (8.2); or

c) the scope unit has a <package use clause> which mentions a <package> such that:
   1) either the <package use clause> has the <definition selection list> omitted or the <name> of the entity is mentioned in a <definition selection>; and
   2) the <package> that is the defining context for the entity either has the <package interface> omitted or <name> of the entity is mentioned in the <package interface>; or

d) the scope unit contains an <interface definition> which is the defining context of the entity (see 12.1.2); or

e) the scope unit contains a <data type definition> which is the defining context of the entity and it is not protected from visibility by a special construction defined in 12.1.9.3; or

f) the entity is visible in the scope unit that defines that scope unit.

It is allowed to omit some of the leftmost <path item>s, or the whole <qualifier> of an <identifier> if the omitted <path item>s can be uniquely expanded to a full <qualifier>.

When the <name> part of an <identifier> denotes an entity that is not of entity kind h), the <name> 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 (resolution by container). If the <identifier> does not contain a <qualifier>, then the requirement on matching of <qualifier>s does not apply.

The binding of a <name> to a definition through resolution by container proceeds in the following steps, starting with the scope unit denoted by the partial <qualifier>:

a) if a unique entity exists in a scope unit with the same <name> and entity kind, the <name> is bound to that entity; otherwise

b) if the scope unit is a specialization, step a) is repeated recursively until the <name> can be bound to an entity; otherwise

c) if the scope unit has a <package use clause> and a unique entity exists and is visible in the <package>, the <name> is bound to that entity; otherwise

d) if the scope unit has an <interface definition> and a unique entity exists and is visible in the <interface definition>, the <name> is bound to that entity; otherwise

e) resolution by container is attempted in the scope unit that defines the current scope unit.

With respect to visibility and use of qualifiers, a <package use clause> associated to a scope unit is regarded as representing a package definition directly enclosing the scope unit and defined in the scope unit where that scope unit is defined. If the <identifier> does not contain a <qualifier>, a <package use clause> is considered as the nearest enclosing scope unit to the scope unit to which it is associated and contains the entities visible from the package.

NOTE 3 – In the concrete syntax, packages cannot be defined inside other scope units. The above rule is only for defining the visibility rules that apply for packages. A consequence of this rule is that names in a package can be referred to using different qualifiers, one for each enclosed <package use clause> of the package.

When the <name> part of an <identifier> denotes an entity of the entity kind h), the binding of the <name> to a definition must be resolvable by context. Resolution by context is attempted after resolution by container; that is, if a <name> can be bound to an entity through resolution by container that binding is used even if resolution by context could bind that <name> to an entity also. The context for resolving a <name> is an <assignment> (if the <name> occurred in an <assignment>), a <decision> (if the <name> occurred in the <question> or <answer>s of a <decision>), or an <expression> which is not part of any other <expression> otherwise. Resolution by context proceeds as follows:

a) For each <name> occurring in the context, find the set of <identifier>s, such that the <name> part is visible, having the same <name> and partial <qualifier> taking renaming into account.

b) Construct the product of the sets of <identifier>s associated with each <name>.

c) Consider only those elements in the product which do not violate any static sort constraints taking into account also those sorts in packages that are not made visible in a <package use clause>. Each remaining element represents a possible, statically correct binding of the <name>s in the <expression> to entities.

d) Due to the possibility of polymorphism in <assignment>s (see 12.3.3), the static sort of an <expression> may not be the same as the static sort of the <variable>, and similarly for the implicit assignments in parameters. The number of such mismatches is counted for each element.

e) Compare the elements in pairs dropping those with more mismatches.

f) If there is more than one remaining element, all non-unique <identifier>s must represent the same *Dynamic-operation-signature*; otherwise the <name>s in the context cannot be bound to a definition.

It is only allowed to omit the optional <scope unit kind> in a <path item> if the <name> or <quoted operation name> uniquely determines the scope unit.

There is no corresponding abstract syntax for the <scope unit kind> denoted by **operator** or **method**.

In the concrete textual grammar, the optional name or identifier in a definition after the ending keywords (**endsystem**, **endblock**, etc.) in definitions must be syntactically the same as the name or identifier following the corresponding commencing keyword (**system**, **block**, etc., respectively).

6.4 Informal text

Abstract grammar

```
Informal-text                    ::      ...
```

Concrete textual grammar

```
<informal text>      ::=
                          <character string>
```

Semantics

If informal text is used in a specification, it means that this text does not have any semantics defined by SDL. The semantics of the informal text can be defined by some other means.

6.5 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, which may cross each other. There is no logical association between symbols that do cross. The following are line symbols:

```
<association symbol>
<channel symbol>
<create line symbol>
<dashed association symbol>
<dependency symbol>
<flow line symbol>
<solid association symbol>
<solid on exception association symbol>
 <specialization relation symbol>
```

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>, <raise symbol>, <handle 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).

6.6 Partitioning of drawings

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

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

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

<heading> ::=
    <kernel heading> [<extra heading>]

<kernel heading> ::=
    [<virtuality>] [**exported**]
    <drawing kind> <drawing qualifier> | <drawing name>

<drawing kind> ::=

package |	**system** [**type**] |	**block** [**type**]	| **process** [**type**]
state [**type**] |	**procedure** |	**operator**	| **method**

<extra heading> ::=
    part of drawing heading not in kernel heading

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

<page number> ::=

<literal name>

<number of pages> ::=

<Natural literal name>

<symbol> ::=

any of the terminals defined with a rule name ending in "symbol"

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

A <symbol> is a graphical non-terminal symbol (see 5.4.3).

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 drawing.

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

<virtuality> denotes the virtuality of the type defined by the drawing (see 8.3.2) and **exported** whether a procedure is exported as a remote procedure (see 10.5).

The drawings of SDL/GR are <specification area>, <package diagram>, <agent diagram>, <agent type diagram>, <procedure diagram>, <operation diagram>, <composite state area>, and <composite state type diagram>.

6.7 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>.

The concrete syntax of the second form is:

<end> ::=

[<comment>] <semicolon>

<comment> ::=

comment <character string>

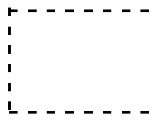
<end> in <package text area>, <agent text area>, <procedure text area>, <composite state text area>, <operation text area>, and <statement list> shall not contain <comment>.

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>.

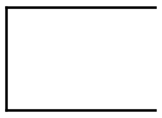
A <comment symbol> can be connected to any graphical symbol by means of a <dashed association symbol>. The <comment symbol> is considered as a closed symbol by completing (in imagination) the rectangle to enclose the text. It contains comment text related to the graphical symbol.

NOTE – <text> in *Concrete graphical grammar* corresponds to <character string> in *Concrete textual grammar* without the enclosing <apostrophe>s. Any conversion between the graphical and textual notations doubles or singles the enclosed <apostrophe>s.

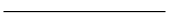
6.8 Text extension

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

<text extension 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 that can contain text by means of a <solid association symbol>. The <text extension symbol> is considered as a closed symbol by completing (in imagination) the rectangle.

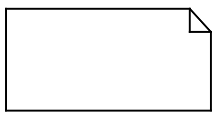
The <text> contained in the <text extension symbol> is a continuation of the text within the graphical symbol and is considered to be contained in that symbol and is therefore treated as a number of lexical units.

6.9 Text symbol

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

Concrete graphical grammar

<text symbol> ::=



7 Organization of SDL specifications

An SDL system cannot usually be described easily as a single independent piece of text or on a single diagram. The language therefore supports the partitioning of the specification and use of SDL from elsewhere.

7.1 Framework

An <sdl specification> can be described as a monolithic <system specification> (possibly augmented by a collection of <package>s) 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, that is, 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 "inserted" into exactly one place (the defining context) using a reference. A

<specification area> allows a graphical depiction of the relationships between <system specification> and <package>s.

Abstract grammar

```
SDL-specification              ::     [ Agent-definition ]
                                      Package-definition-set
```

Concrete grammar

```
<sdl specification> ::=
                        [<specification area>]
                        { <package> | <system specification> } <package>* <referenced definition>*

<system specification> ::=
                        <textual system specification> | <graphical system specification>

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

Concrete textual grammar

```
<textual system specification> ::=
                        <agent definition>
                    |   <textual typebased agent definition>
```

Concrete graphical grammar

```
<graphical system specification> ::=
                        <agent diagram>
                    |   <graphical typebased agent definition>

<specification area> ::=
                        <frame symbol> contains
                    {   <agent reference area>
                    |   <graphical typebased agent definition>
                        [     is connected to {<graphical package use area>+ }set]
                    }
                        {<package reference area>* }set}
```

Semantics

An *SDL-specification* has the combined semantics of the system agent (if one is given) with the packages. If no system agent is specified, the specification provides a set of definitions for use in other specifications.

Model

A <system specification> being a <process definition> or a <textual typebased process definition> is derived syntax for a <system definition> having the same name as the process, containing implicit channels and containing the <process definition> or <textual typebased process definition> as the only definition.

A <system specification> being a <process diagram> or a <graphical typebased process definition> is derived syntax for a <system diagram> having the same name as the process, containing implicit channels and containing the <process diagram> or <graphical typebased process definition> as the only definition.

A <system specification> being a <block definition> or a <textual typebased block definition> is derived syntax for a <system definition> having the same name as the block, containing implicit channels and containing the <block definition> or <textual typebased block definition> as the only definition.

A <system specification> being a <block diagram> or a <graphical typebased block definition> is derived syntax for a <system diagram> having the same name as the block, containing implicit channels and containing the <block diagram> or <graphical typebased block definition> as the only definition.

7.2 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, signal lists, remote specifications and synonyms.

Definitions within a package are made visible to another scope unit by a package use clause.

Abstract grammar

Package-definition	::	*Package-name*
		*Package-definition-**set***
		*Data-type-definition-**set***
		*Syntype-definition-**set***
		*Signal-definition-**set***
		*Exception-definition-**set***
		*Agent-type-definition-**set***
		*Composite-state-type-definition-**set***
		*Procedure-definition-**set***

Concrete textual grammar

```
<package definition> ::=
                        {<package use clause>}*
                         <package heading> <end>
                             {<entity in package>}*
                        endpackage [<package name>] <end>

<package heading> ::=
                        package [ <qualifier> ] <package name>
                        [<package interface>]

<entity in package> ::=
                        <agent type definition>
                   |    <agent type reference>
                   |    <package definition>
                   |    <package reference>
                   |    <signal definition>
                   |    <signal reference>
                   |    <signal list definition>
                   |    <remote variable definition>
                   |    <data definition>
                   |    <data type reference>
                   |    <procedure definition>
                   |    <procedure reference>
                   |    <remote procedure definition>
                   |    <composite state type definition>
                   |    <composite state type reference>
                   |    <exception definition>
                   |    <select definition>
                   |    <macro definition>
                   |    <interface reference>
                   |    <association>

<package reference> ::=
                        package [ <qualifier> ] <package name> referenced <end>

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

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

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

<selected entity kind> ::=
system type
| **block type**
| **process type**
| **package**
| **signal**
| **procedure**
| **remote procedure**
| **type**
| **signallist**
| **state type**
| **synonym**
| **remote**
| **exception**
| **interface**

<package interface> ::=
public <definition selection list>

For each <package identifier> mentioned in a <package use clause>, there must exist a corresponding <package>. This package may be part of <sdl specification> or may be a package contained in another package or else there must exist a mechanism for accessing the referenced <package>, just as if it were a part of the <sdl specification>. This mechanism is not defined in this Recommendation.

If the package is part of <sdl specification> or if there exists a mechanism for accessing the reference <package> there must not be a <qualifier> in <package identifier>.

If the corresponding <package> is contained in another package, the <package identifier> reflects the hierarchical structure from the outermost <package> to the defined <package>. Leftmost <path item>s can be omitted.

The <package identifier> must denote a visible package. All <package>s in the <qualifier> of the fully qualified <package identifier> must be visible. A package is visible if it is either part of the <sdl specification> or if its <identifier> is visible according to the visibility rules of SDL for <identifier>. The visibility rules of SDL imply that a <package identifier> can be made visible with a <package use clause> and that a package is visible in the scope in which it is contained. This scope extends also to the <package use clause> of the container package.

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

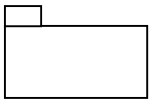
The <selected entity kind> **procedure** is used for selection of both (normal) procedures and remote procedures. If both a normal procedure and a remote procedure have the given <name>, **procedure** denotes the normal procedure. To force the <definition selection> to denote the remote procedure, the **procedure** keyword can be preceded by **remote**.

The keyword **type** is used for selection of a sort name and also a syntype name in a package. **remote** is used for selection of a remote variable definition.

Concrete graphical grammar

<package diagram> ::=
<frame symbol> ***contains***
{ <package heading>
{ {<package text area>}*
{<diagram in package>}* } ***set*** }
[***is associated with*** <package use area>]

```
<package use area> ::=
                    <text symbol> contains {<package use clause>}*

<package text area> ::=
                    <text symbol> contains
                         {    <agent type reference>
                         |    <package reference>
                         |    <signal definition>
                         |    <signal reference>
                         |    <signal list definition>
                         |    <remote variable definition>
                         |    <data definition>
                         |    <data type reference>
                         |    <procedure definition>
                         |    <procedure reference>
                         |    <remote procedure definition>
                         |    <exception definition>
                         |    <select definition>
                         |    <macro definition>
                         |    <interface reference> }*

<diagram in package> ::=
                    <package diagram>
               |    <package reference area>
               |    <type in agent area>
               |    <data type reference area>
               |    <signal reference area>
               |    <procedure reference area>
               |    <interface reference area>
               |    <create line area>
               |    <option area>

<package reference area> ::=
                    <package symbol> contains <package identifier>
                         [    is connected to {<graphical package use area>+ }set]

<graphical package use area> ::=
                    <dependency symbol> is connected to { <package diagram> | <package reference area> }

<package symbol> ::=
```



```
<dependency symbol> ::=
                    ----------->
```

The <package use area> must be placed on the top of the <frame symbol>. The optional <qualifier> and <package name> of a <package reference area> must be contained in the lower rectangle of <package symbol>.

The <graphical package use area>s for a <package reference area> are partial specifications of the corresponding <package use clause> for the <package diagram> or <package> or <system specification> for a <package reference area> in a <specification area>, and must be consistent with this <package use clause>.

Semantics

The visibility of the name of an entity defined within a <package> is explained in 6.3.

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 an agent.

If a name in a <definition selection> denotes a <sort>, the <definition selection> also implicitly denotes the data type that defined the <sort> and all the literals and operations defined by the data type. If a name in a <definition selection> denotes a syntype, the <definition selection> also implicitly denotes the data type that defined the <parent sort identifier> and all the literals and operations defined by the data type.

The <selected entity kind> in <definition selection> denotes the entity kind of the <name>. Any pair of (<selected entity kind>, <name>) must be distinct within a <definition selection list>. For a <definition selection> in an <package interface>, the <selected 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 use clause>, <selected 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.

Model

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

```
        use Predefined;
```

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

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

7.3 Referenced definition

Concrete grammar

```
<referenced definition> ::=
                        <definition> | <diagram>
```

Concrete textual grammar

```
<definition> ::=
                        <package definition>
                |       <agent definition>
                |       <agent type definition>
                |       <composite state>
                |       <composite state type definition>
                |       <procedure definition>
                |       <operation definition>
                |       <macro definition>
```

Concrete graphical grammar

```
<diagram> ::=
                        <package diagram>
                |       <agent diagram>
                |       <agent type diagram>
                |       <composite state area>
                |       <composite state type diagram>
                |       <procedure diagram>
                |       <operation diagram>
```

For each <referenced definition>, except for <macro definition>, there must be a reference in the associated <package> or <system specification>. Textual and graphical references are defined as <... reference> and <… reference area> respectively (for example <block reference> and <block reference area>).

An optional <qualifier> and <name> 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>, the <qualifier> of one must not constitute the leftmost part of the other <qualifier>, and neither <qualifier> can be omitted. The <qualifier> must be present if the <referenced definition> is a <package definition>.

It is not allowed to specify a <qualifier> after the initial keyword(s) for definitions which are not <referenced definition>s.

Model

Before the properties of a <system specification> are derived, each reference is replaced by the corresponding <referenced definition>. In this replacement, the <qualifier> of the <referenced definition> is removed. If the <referenced definition> is a <diagram> referenced from a <definition>, or is <definition> referenced from a <diagram>, the <referenced definition> is considered translated to the appropriate grammar during the replacement.

8 Structural Concepts

This clause introduces a number of language mechanisms to support the modelling of application specific phenomena by instances and application specific concepts by types. Inheritance is intended to represent concept generalization and 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 the 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.

8.1 Types, instances and gates

There is a distinction between definition of instances (or set of instances) and definition of types in SDL descriptions. This clause introduces (in 8.1.1) type definitions for agents, and (in 8.1.3) corresponding instance specifications, while the introduction of other types are in procedures (9.4), signals (10.3), timers (11.15), sorts (12.1.1) and interfaces (12.1.2). An agent type definition is not connected (by channels) to any instances; instead, agent type definitions introduce gates (8.1.6). These are connection points on the typebased instances for channels.

A type defines a set of properties. All instances of the type (5.2.1) have this set of properties.

An instance (or instance set) always has a type, which is implied if the instance is not explicitly based on a type. For example, a process diagram has an implied equivalent anonymous process type.

8.1.1 Structural type definitions

These are type definitions for entities that are used in the structure of a specification. In contrast, procedure definitions are also type definitions, but organize behaviour rather than structure.

8.1.1.1 Agent types

An agent type is a system, block or process type. When the type is used to define an agent, the agent is of corresponding kind (system, block or process).

Abstract grammar

```
Agent-type-definition          ::   Agent-type-name
                                    Agent-kind
                                    [ Agent-type-identifier ]
                                    Agent-formal-parameter*
                                    Data-type-definition-set
                                    Syntype-definition-set
                                    Signal-definition-set
                                    Timer-definition-set
                                    Exception-definition-set
                                    Variable-definition-set
                                    Agent-type-definition-set
                                    Composite-state-type-definition-set
                                    Procedure-definition-set
                                    Agent-definition-set
                                    Gate-definition-set
                                    Channel-definition-set
                                    [ State-machine-definition ]
Agent-type-identifier          =    Identifier
```

Concrete textual grammar

```
<agent type definition> ::=
                         <system type definition> | <block type definition> | <process type definition>

<agent type structure> ::=
                         [<valid input signal set>]
                         {    {    <entity in agent>
                              |    <channel definition>
                              |    <gate in definition>
                              |    <agent definition>
                              |    <agent reference>
                              |    <textual typebased agent definition> }*
                              [ <state partitioning>]
                         |    {    <entity in agent>
                              |    <gate in definition> }*
                               <agent type body> }

<type preamble> ::=
                         [ <virtuality> | <abstract> ]

<agent type additional heading> ::=
                         [<formal context parameters>] [<virtuality constraint>]
                         <agent additional heading>

<agent type reference> ::=
                         <system type reference>
                    |    <block type reference>
                    |    <process type reference>

<agent type body> ::=
                         [ [<on exception>] <start> ]
                         { <state> | <exception handler> | <free action> } *
```

Concrete graphical grammar

```
<agent type diagram> ::=
                         <system type diagram> | <block type diagram> | <process type diagram>
```

```
<type in agent area> ::=
                         <agent type diagram>
                    |    <agent type reference area>
                    |    <composite state type diagram>
                    |    <composite state type reference area>
                    |    <procedure diagram>
                    |    <procedure reference area>
                    |    <data type reference area>
                    |    <signal reference area>
                    |    <association area>

<agent type diagram content> ::=
                    {    <agent text area>*
                         <type in agent area>*
                         { <interaction area> | <agent type body area> } }set

<agent type body area> ::=
                         <type state machine graph area>

<type state machine graph area> ::=
                    {    [<on exception association area>] [<start area>]
                         { <state area> | <exception handler area> | <in connector area> }* }set

<agent type reference area> ::=
                         {       <system type reference area>
                                | <block type reference area>
                                | <process type reference area> }
                         is connected to { <gate property area>* }set
```

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

If there is an <agent type reference area> for the agent defined by an <agent type diagram>, the <gate property area>s associated with the <agent type reference area> correspond to the <gate on diagram>s associated with the <agent type diagram>. No <gate property area> associated with the <agent type reference area> must contain <signal list item>s not contained in the corresponding <gate on diagram>s associated with the <agent type diagram>.

Semantics

An *Agent-type-definition* defines an agent type. All agents of an agent type have the same properties as defined for that agent type.

Signals mentioned in <output>s of the state machine of an agent type must be in the complete valid input signal set of the agent type or in the <signal list> of a gate in the direction from the agent type.

The definition of an agent type implies the definition of an interface in the same scope of the agent type (see 12.1.2). The pid sort implicitly defined by this interface is identified with *Agent-type-name* and is visible in the same scope unit as where the agent type is defined.

NOTE – Because every agent type has an implicitly defined interface with the same name, the agent type must have a different name from every explicitly defined interface, and every agent (these also have implicit interfaces) defined in the same scope, otherwise there are name clashes.

The properties defined in an *Agent-type-definition* such as the *Procedure-definition-**set***, *Agent-definition-**set**,* and *Gate-definition-**set*** determine the properties of any *Agent-definition* based on the type, and are therefore described in clause 9.

Model

An agent type with an <agent type body> or an <agent type body area> is shorthand for an agent type having only a state machine, but no contained agents. This state machine is obtained by replacing the <agent type body> or <agent type body area> by a composite state definition. This composite state definition has the same name as the agent type and its *State-transition-graph* is represented by the <agent type body> or the <agent type body area>.

8.1.1.2 System type

A system type definition is a top-level agent type definition. It is denoted by the keyword **system type**. A system type definition must not be contained in any other agent or agent type definition. A system type can neither be abstract nor virtual.

Concrete textual grammar

<system type definition> ::=
    <package use clause>*
    <system type heading> <end> <agent type structure>
    endsystem type [[<qualifier>] <system type name>] <end>

<system type heading> ::=
    system type [<qualifier>] <system type name>
        <agent type additional heading>

<system type reference> ::=
    system type <system type identifier> <type reference properties>

A <formal context parameter> of <formal context parameters> must not be an <agent context parameter>, <variable context parameter> or <timer context parameter>.

The <agent type additional heading> in a <system type definition> or a <system type diagram> may not include <agent formal parameters>.

Concrete graphical grammar

<system type diagram> ::=
    <frame symbol> ***contains*** {<system type heading> <agent type diagram content> }
    [***is associated with*** <package use area>]

<system type reference area> ::=
    <type reference area>

<system type symbol> ::=
    <block type symbol>

The <type reference area> that is part of a <system type reference area> must have a <graphical type reference heading> that contains a <system type name>.

Semantics

A <system type definition> defines a system type.

8.1.1.3 Block type

Concrete textual grammar

<block type definition> ::=
    <package use clause>*
    <block type heading> <end> <agent type structure>
    endblock type [[<qualifier>] <block type name>] <end>

<block type heading> ::=
    <type preamble>
    block type [<qualifier>] <block type name>
        <agent type additional heading>

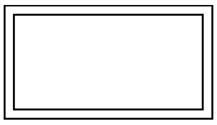
<block type reference> ::=
    <type preamble>
    block type <block type identifier> <type reference properties>

Concrete graphical grammar

<block type diagram> ::=
    <frame symbol> ***contains*** {<block type heading> <agent type diagram content> }
    is connected to {{ <gate on diagram>* }***set*** }
    [***is associated with*** <package use area>]

<block type reference area> ::=
<type reference area>

<block type symbol> ::=



The <type reference area> that is part of a <block type reference area> must have a <graphical type reference heading> that contains a <block type name>.

The <gate on diagram> in a <block type diagram> may not include <external channel identifiers>.

The < gate on diagram>s in a <block type diagram> must be outside the diagram frame.

Semantics

A <block type definition> defines a block type.

8.1.1.4 Process type

Concrete textual grammar

<process type definition> ::=
<package use clause>*
<process type heading> <end> <agent type structure>
endprocess type [[<qualifier>] <process type name>] <end>

<process type heading> ::=
<type preamble>
process type [<qualifier>] <process type name>
<agent type additional heading>

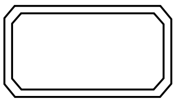
<process type reference> ::=
<type preamble>
process type <process type identifier> <type reference properties>

Concrete graphical grammar

<process type diagram> ::=
<frame symbol> ***contains*** {<process type heading> <agent type diagram content> }
is connected to {{ <gate on diagram>* }***set*** }
[***is associated with*** <package use area>]

<process type reference area> ::=
<type reference area>

<process type symbol> ::=



The <type reference area> that is part of a <process type reference area> must have a <graphical type reference heading> that contains a <process type name>.

Semantics

A <process type definition> defines a process type.

The <gate on diagram> in a <process type diagram> may not include <external channel identifiers>.

The <gate on diagram>s in a <process type diagram> must be outside the diagram frame.

8.1.1.5 Composite state type

Abstract grammar

```
Composite-state-type-definition      ::   State-type-name
                                          [ Composite-state-type-identifier ]
                                          Composite-state-formal-parameter*
                                          State-entry-point-definition-set
                                          State-exit-point-definition-set
                                          Gate-definition-set
                                          Connect-node-set
                                          Data-type-definition-set
                                          Syntype-definition-set
                                          Exception-definition-set
                                          Composite-state-type-definition-set
                                          Variable-definition-set
                                          Procedure-definition-set
                                          [ Composite-state-graph | State-aggregation-node ]
Composite-state-type-identifier      ::   Identifier
```

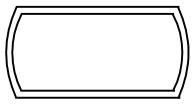
The *Gate-definition-set* must be empty unless the composite state is used as a *State-machine-definition*.

Concrete textual grammar

```
<composite state type definition> ::=
                        {<package use clause>}*
                        <composite state type heading> <end> substructure
                              [<valid input signal set>]
                              {<gate in definition>}*
                              {<state connection points>}*
                              {<entity in composite state>}*
                              <composite state body>
                        endsubstructure state type [ [ <qualifier> ] <composite state type name> ] <end>
                  |     {<package use clause>}*
                        <state aggregation type heading> <end> substructure
                              [<valid input signal set>]
                              {<gate in definition>}*
                              {<state connection points>}*
                              {<entity in composite state>}*
                              <state aggregation body>
                        endsubstructure state type [ [ <qualifier> ] <composite state type name> ] <end>

<composite state type heading> ::=
                        [<virtuality>]
                        state type [ <qualifier> ] <composite state type name>
                              [<formal context parameters>] [<virtuality constraint>]
                              [<specialization>]
                              [<agent formal parameters>]

<state aggregation type heading> ::=
                        [<virtuality>]
                        state aggregation type [ <qualifier> ] <composite state type name>
                              [<formal context parameters>] [<virtuality constraint>]
                              [<specialization>]
                              [<agent formal parameters>]

<composite state type reference> ::=
                        <type preamble>
                        state type <composite state type identifier> <type reference properties>
```

Concrete graphical grammar

```
<composite state type diagram> ::=
                    <frame symbol>
                    contains {
                         <composite state type heading>
                         {      {<composite state text area>}*
                                {<type in composite state area>}*
                                <composite state body area> }set }
                    is associated with { <graphical state connection point>* }set
                    is connected to {{ <gate on diagram>* }set }
                    [ is associated with <package use area> ]
               |    <frame symbol>
                    contains {
                         <state aggregation type heading>
                         {      {<composite state text area>}*
                                {<type in composite state area>}*
                                <state aggregation body area> }set }
                    is associated with { <graphical state connection point>* }set
                    is connected to {{ <gate on diagram>* }set }
                    [ is associated with <package use area> ]

<composite state type reference area> ::=
                    <type reference area>
                    is connected to {<gate property area>*}set

<composite state type symbol> ::=
```



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

The <type reference area> that is part of a <composite state type reference area> must have a <graphical type reference heading> that contains a <composite state type name>.

The <gate on diagram> in a <composite state type diagram> may not include <external channel identifiers>.

The <gate on diagram>s in a <composite state type diagram> must be outside the diagram frame.

If there is a <composite state type reference area> for a composite state defined by a <composite state type diagram>, the <gate property area>s associated with the <composite state type reference area> correspond to the <gate on diagram>s associated with the <composite state type diagram>. No <gate property area> associated with the <composite state type reference area> must contain <signal list item>s not contained in the corresponding <gate on diagram> associated with the <composite state type diagram>.

Semantics

A *Composite-state-type-definition* defines a composite state type. All composite states of a composite state type have the same properties as defined for that composite state type. The semantics are further defined in 11.11.

8.1.2 Type expression

A type expression is used for defining one type in terms of another as defined by specialization in 8.3.

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 8.2.

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 there are 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.

In addition to fulfilling any static conditions on the definition denoted by the <base type>, usage of the <type expression> must also fulfil any static condition on the resultant type.

NOTE – The static properties on the usage of a <type expression> may be violated in the following cases, for example:

– When a scope unit has signal context parameters or timer context parameters, the condition that stimuli for a state must be disjoint, depends on the actual context parameters that will be used.

– When an output in a scope unit refers to a gate or a channel, which is not defined in the nearest enclosing type having gates, instantiation of that type results in an erroneous specification if there is no communication path to the gate.

– When a procedure contains references to signal identifiers, remote variables and remote procedures, specialization of that procedure inside an agent results in an erroneous specification if the usage of such identifiers inside the procedure violates valid usage for the process.

– When state types are instantiated as parts of the same state aggregation, the resulting composite state is erroneous if two or more parts have the same signal in the input signal set.

– When a scope unit has an agent context parameter that is used in an output action, the existence of a possible communication path depends on which actual context parameter will be used.

– When a scope unit has a sort context parameter, application of an actual sort context parameter will result in an erroneous specification if a polymorphic assignment to a value is attempted in the specialized type.

– If a formal parameter of a procedure added in a specialization has the <parameter kind> in/out or out, a call in the supertype to a subtype (using this) will result in an omitted actual in/out or out parameter, that is, in an erroneous specification.

– 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 or out, a call of the formal procedure context parameter in the parameterized type may result in an omitted actual in/out or out parameter, that is, in 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 as <actual context parameter>s. These <actual context parameter>s now have the defining context in the current scope unit.

8.1.3 Definitions based on types

A typebased agent definition defines an agent instance set according to a type denoted by <type expression>. The defined entities get the properties of the types that they are based on.

Concrete textual grammar

```
<textual typebased agent definition> ::=
                              <textual typebased system definition>
                        |     <textual typebased block definition>
                        |     <textual typebased process definition>
```

The agent type denoted by <base type> in the type expression of a <textual typebased agent definition> must contain an unlabelled start transition in its state machine.

Concrete graphical grammar

```
<graphical typebased agent definition> ::=
                              <graphical typebased system definition>
                        |     <graphical typebased block definition>
                        |     <graphical typebased process definition>

<inherited agent definition> ::=
                              <inherited block definition>
                        |     <inherited process definition>
```

The agent type denoted by <base type> in the type expression of a <graphical typebased agent definition> or <inherited agent definition> must contain an unlabelled start transition in its state machine.

In a <graphical typebased agent definition>, <graphical gate definition>s and <graphical interface gate definition>s must be placed outside the <block symbol> or <process symbol>.

8.1.3.1 System definition based on system type

Concrete textual grammar

```
<textual typebased system definition> ::=
                              <typebased system heading> <end>

<typebased system heading> ::=
                              system <system name> <colon> <system type expression>
```

Concrete graphical grammar

```
<graphical typebased system definition> ::=
                              <block symbol> contains <typebased system heading>
```

Semantics

A typebased system definition defines an *Agent-definition* with *Agent-kind* **SYSTEM** 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>.

8.1.3.2 Block definition based on block type

Concrete textual grammar

<textual typebased block definition> ::=
    block <typebased block heading> <end>

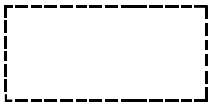
<typebased block heading> ::=
    <block name> [<number of instances>] <colon> <block type expression>

Concrete graphical grammar

<graphical typebased block definition> ::=
    <block symbol> ***contains*** { <typebased block heading> { <gate>* }***set*** }
    is connected to { {<gate property area>*}***set*** }

<inherited block definition> ::=
    <dashed block symbol> ***contains*** { <block identifier> { <gate>* }***set*** }
    is connected to { {<gate property area>*}***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 *Agent-definition*s of *Agent-kind* **BLOCK** 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>.

An <inherited 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 inherited block may be specified.

8.1.3.3 Process definition based on process type

Concrete textual grammar

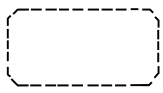
<textual typebased process definition> ::=
    process <typebased process heading> <end>

<typebased process heading> ::=
    <process name> [<number of instances>] <colon> <process type expression>

Concrete graphical grammar

<graphical typebased process definition> ::=
    <process symbol> ***contains*** { <typebased process heading> { <gate>* }***set*** }
    is connected to { {<gate property area>*}***set*** }

<inherited process definition> ::=
<dashed process symbol> ***contains*** { <process identifier> { <gate>* }***set*** }
is connected to { {<gate property area>*}***set*** }

<dashed process symbol> ::=


The <gate>s are placed near the border of the symbols and associated with the connection point to channels.

Semantics

A typebased process definition defines an *Agent-definition* with *Agent-kind* **PROCESS** derived by transformation from a process type.

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 <inherited process definition> can only appear in a subtype definition. It represents the process defined in the supertype of the subtype definition. Additional channels connected to gates of the inherited process may be specified.

8.1.3.4 Composite state definition based on composite state type

Concrete textual grammar

<textual typebased state partition definition> ::=
state aggregation <typebased state partition heading> <end>

<typebased composite state> ::=
<state name> [<actual parameters>] <colon> <composite state type expression>

Semantics

A typebased composite state definition and a textual typebased state partition definition define a composite state derived by transformation from a composite state type.

Model

A <typebased composite state> is transformed to a <composite state> which has the definitions of the composite state type as defined by <composite state type expression>.

8.1.4 Abstract type

Concrete grammar

<abstract> ::=
abstract

<abstract> is part of the type definition. See 8.1.1.1, 8.1.5, 9.4, 10.3 and 12.1.1.

A type is an abstract type if its definition contains <abstract>.

An abstract procedure cannot be called.

A typebased agent (see 8.1.3) shall not be specified with an abstract agent type as the type.

An <operation identifier> must not represent the *Operation-identifier* for the Make operator defined for an abstract data type.

A <literal identifier> must not represent a *Literal-identifier* denoting a *Literal-signature* of an abstract data type.

An abstract type cannot be instantiated. However, a subtype of an abstract data type can be instantiated, if it is not itself abstract.

8.1.5 Type references

Type diagrams and entity type definitions may have type references. A type reference specifies both that a type is defined in the scope unit of the containing definition or diagram (but fully described in the referenced definition or diagram), and that this type has the properties being partially specified as part of the type reference. The referenced definition or diagram defines the properties of the type, while the type references are only partial definitions. It is required that the partial specification as part of a type reference is consistent with the specification of the type definition or diagram. A partial specification of a variable, for example, may give the variable name but not the sort of the variable. There must be a variable of that name in the referenced definition, and in this definition a sort must be specified.

The same type definition may have several type references. The references without a qualifier must all be in the same scope unit, and the type definition is inserted in this scope unit.

Concrete textual grammar

```
<type reference properties> ::=
                    [ with { <attribute property> <end> }+ ]
                    [ with { <behaviour property> <end> }+ ]
                    referenced <end>
```

If <type reference properties> appears in <package text area>, <agent text area>, <procedure text area>, <composite state text area>, and <operation text area>, neither <attribute property>s nor <behaviour property>s must be present.

```
<attribute property> ::=
                    <variable property>
               |    <field property>
               |    <signal parameter property>
               |    <interface variable property>
```

An <attribute property> provides a partial specification of properties of variables or fields being defined in the type definition that is referenced by the type reference. The elements of <attribute property> must be consistent with the corresponding properties in the referenced type definition.

```
<variable property> ::=
                    [ <local> | <exported> ] <variable name> [<sort>]

<local> ::=
                    <hyphen>

<exported> ::=
                    <plus sign>
```

A <variable property> corresponds to a <variable definition> in an agent type, procedure or composite state type. <local> indicates a local variable; <exported> indicates an exported variable. <variable name> and <sort>, if present, must be the same as in the corresponding variable definition.

```
<field property> ::=
                    [<symbolic visibility>] <field name> [<sort>]

<symbolic visibility> ::=
                    <private>
               |    <public>
               |    <protected>

<private> ::=
                    <hyphen> | private

<public> ::=
                    <plus sign> | public
```

<protected> ::=

<number sign> | **protected**

A <field property> corresponds to a <field> in a data type. <private> (<public>, <protected>) corresponds to private (public, protected) <visibility> in the corresponding field. <field name> and <sort>, if present, must be the same as in the corresponding field definition.

<signal parameter property> ::=

<sort list>

A <signal parameter property> corresponds to a list of signal parameters in a signal definition. The <sort list> must correspond to the <sort list> in <signal definition item> of the corresponding signal definition. There must be only one <signal parameter property> in the <type reference properties> or <type reference area>.

<interface variable property> ::=

<remote variable name> [<sort>]

An <interface variable property> corresponds to an <interface variable definition> in an interface. The <sort> must be the same as the <sort> in the interface variable definition.

<behaviour property> ::=

```
        { [operator] <operation property> }
|       { [method] <operation property> }
|       { [procedure] <procedure property> }
|       { [signal] <signal property> }
|       { [exception] <exception property> }
|       { [timer] <timer property> }
|       { <interface use list> }
```

A <behaviour property> provides a partial specification of properties of procedures and operations being defined in the type definition that is referenced by the type reference, and this specification must be consistent with the corresponding definitions in the corresponding type definition.

<operation property> ::=

```
[<symbolic visibility>] <operation name>
        <procedure signature>
```

An <operation property> corresponds to an <operation definition> in an object or value type reference. <private> (<public>, <protected>) corresponds to private (public, protected) <visibility> in the corresponding operation definition. The list of <formal parameter>s, <result>, and <raises> in <procedure signature>, if present, must be the same as the <formal parameter>s, <result>, and <raises>, respectively, in the corresponding operation definition.

<procedure property> ::=

```
[ <local> | <exported> ] <procedure name>
        <procedure signature>
```

A <procedure property> in an agent type reference corresponds to a <procedure definition> in the agent type. <local> indicates a local procedure; <exported> indicates an exported procedure. The list of <formal parameter>s, <result>, and <raises> in <procedure signature>, if present, must be the same as the <procedure formal parameter>s, <procedure result>, and <raises>, respectively, in the corresponding procedure definition.

A <procedure property> in an interface reference corresponds to an <interface procedure definition> in an interface. <local> must not be present in an interface reference. <procedure signature>, if present, must be the same as in the corresponding interface procedure definition.

<signal property> ::=

<signal name> [<sort list>]

A <signal property> in an agent type reference corresponds to a signal handled in an input in the agent type.

A signal property in an interface reference corresponds to a <signal definition item> in a <signal definition> in the <interface definition>. The <sort list>, if present, must be the same as in the corresponding <signal definition item>.

<exception property> ::=
    <exception name> [<sort list>]

An <exception property> in a type reference corresponds to an <exception definition item> in the type definition being referenced by the type reference. The <sort list>, if present, must be the same as in the corresponding <exception definition item>.

<timer property> ::=
    <timer name> [<sort list>]

A <timer property> in a type reference corresponds to a <timer definition item> in the type definition being referenced by the type reference. The <sort list>, if present, must be the same as in the corresponding <timer definition item>.

An <interface use list> corresponds to an <interface use list> of the interface definition being referenced by the type reference. Each <signal list item> must correspond to a <signal list item> in the <interface use list> of the referenced interface definition.

Concrete graphical grammar

<type reference area> ::=
    { <basic type reference area> | <iconized type reference area> }
    is connected to { <graphical package use area>* <specialization area>* } ***set***

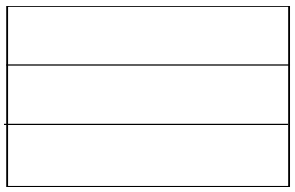
The <graphical package use area> for a <type reference area> is a partial specification of the corresponding <package use clause> for the type diagram, and must be consistent with this.

The <specialization area> shall be connected to the upper part of the <basic type reference area>, or <iconized type reference area>, using the end of the <specialization relation symbol> that has no arrow. There must only be one <specialization area> for all <type reference area>s except an interface reference.

The <specialization area> corresponds to the <specialization> of the referenced type. The connected <type reference area> must correspond to the <base type> in the <type expression> of the <specialization>. The <actual context parameters> in the <specialization area> must correspond to the <actual context parameters> in the <type expression>.

<basic type reference area> ::=
    <class symbol>
    contains {
        <graphical type reference heading>
        <attribute properties area>
        <behaviour properties area> }

<class symbol> ::=



The relative positioning of the two lines dividing the <class symbol> into three compartments is allowed to be different than shown.

<graphical type reference heading> ::=
{ <type reference kind symbol> ***contains*** **system** | <system type symbol> }
<system type type reference heading>
| { <type reference kind symbol> ***contains*** **block** | <block type symbol> }
<block type type reference heading>
| { <type reference kind symbol> ***contains*** **process** | <process type symbol> }
<process type type reference heading>
| { <type reference kind symbol> ***contains*** **state** | <composite state type symbol> }
<composite state type type reference heading>
| { <type reference kind symbol> ***contains*** **procedure** | <procedure symbol> }
<procedure type reference heading>
| <type reference kind symbol> ***contains*** **signal**
<signal type reference heading>
| { <type reference kind symbol> ***contains*** { **value** | **object** } | <data symbol> }
<data type type reference heading>
| { <type reference kind symbol> ***contains*** **interface** | <data symbol> }
<interface type reference heading>

The <graphical type reference heading> shall be placed in the uppermost compartment of the containing <class symbol>.

<type reference heading> ::=
<type preamble> [<qualifier>] <name> [<formal context parameters>]

<type preamble> must correspond to <type preamble> of the referenced type, if the reference is virtual, the referenced type must be virtual. If the reference is abstract the referenced type must be abstract.

The <formal context parameters> corresponds to <formal context parameters> of the referenced type. The <formal context parameter list> must correspond to the <formal context parameter list> of the referenced type.

<type reference kind symbol> ::=
« »

The <type reference kind symbol> is placed above or to the left of the <type reference heading>.

<data symbol> ::=

NOTE 1 – The <data symbol> is a rectangle without any visible frame. This implies that a <graphical type reference heading> not containing a <type reference kind symbol> actually contains a <data symbol>.

The <data symbol> corresponds to a <data type definition> or an <interface definition>.

If the <graphical type reference heading> contains a symbol other than a <type reference kind symbol>, this symbol must be placed in the upper, right corner of the <graphical type reference heading>.

<iconized type reference area> ::=
<system type symbol> ***contains*** <system type type reference heading>
| <block type symbol> ***contains*** <block type type reference heading>
| <process type symbol> ***contains*** <process type type reference heading>
| <composite state type symbol> ***contains*** <composite state type type reference heading>
| <procedure symbol> ***contains*** <procedure type reference heading>

NOTE 2 – There is no <iconized type reference area> corresponding to signals, interfaces, as well as object and value types.

<attribute properties area> ::=
{ { <attribute property> <end> }* }***set***

The first <attribute property> in an <attribute properties area> must be placed uppermost in the middle compartment of the containing <class symbol>. Each subsequent <attribute property> must be placed below the previous one.

<behaviour properties area> ::=
{ { <behaviour property> <end> }* }***set***

The first <behaviour property> in a <behaviour properties area> must be placed uppermost in the lower compartment of the containing <class symbol>. Each subsequent <behaviour property> must be placed below the previous one.

Model

A <type reference heading> without a <qualifier> before the <name> is derived syntax in which the entity identified by the <qualifier> is the enclosing context.

A type reference in which the entity identified by the <qualifier> of the <type reference heading> is different from the enclosing context is considered moved to the context given by the qualifier and therefore the visibility rules of that context apply.

Multiple type references in the same context that refer to the same entity class and have the same qualifier and the same name are equivalent to one type reference from that context with all <attribute property> and <behaviour property> elements of all the references.

After reducing multiple type references, the type reference, in which the <qualifier> of the <type reference heading> is the same as the enclosing context, is replaced by the referenced type as defined in 7.3.

NOTE 3 – The model for type references, in which the entity identified by the <qualifier> of the <type reference heading> is different from the enclosing context, means that the referenced type can be an otherwise invisible type within a scope directly within the enclosing context.

8.1.6 Gate

Gates are defined in agent types (block types, process types) or state types and represent connection points for channels, connecting instances of these types (as defined in 8.1.3) with other instances or with the enclosing frame symbol.

It is possible also to define gates in agents and composite states and this represents a notation for specifying that the considered entity has a named connection point.

Abstract grammar

Gate-definition	::	*Gate-name*
		*In-signal-identifier**-set***
		*Out-signal-identifier**-set***
Gate-name	=	*Name*
In-signal-identifier	=	*Signal-identifier*
Out-signal-identifier	=	*Signal-identifier*

Concrete textual grammar

<gate in definition> ::=
<textual gate definition> | <textual interface gate definition>

<textual gate definition> ::=
gate <gate> [**adding**] <gate constraint> [<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>

<textual interface gate definition> ::=
gate { **in** | **out** } **with** <interface identifier> <end>

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.

An *In-signal-identifier* represents an element in the <signal list> to the gate. An *Out-signal-identifier* represents an element in the <signal list> from the gate.

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 connected to a gate must be compatible with the gate constraint. A channel is compatible with a gate constraint if the other endpoint of the channel is an agent or state of the type denoted by <identifier> in the endpoint constraint or a subtype of this type (in case it contains a <textual endpoint constraint> with **atleast**), and if the set of signals (if specified) on the channel is equal to or is a subset of the set of signals specified for the gate in the respective direction.

If the type denoted by <base type> in a <textual typebased block definition> or <textual typebased process definition> contains channels, the following rule applies: for each combination of (gate, signal, direction) defined by the type, the type must contain at least one channel that – for the given direction – mentions **env** and the gate and either mentions the signal or has no explicit <signal list> associated. In the latter case, it must be possible to derive that the channel is able to carry the signal in the given direction.

If the type contains channels mentioning remote procedures or remote variables, a similar rule applies.

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.

The <interface identifier> of an <textual interface gate definition> must not identify the interface implicitly defined by the entity to which the gate is connected (see 12.1.2).

Concrete graphical grammar

<gate on diagram> ::=
<graphical gate definition> | <graphical interface gate definition>

<gate property area> ::=
<graphical gate definition> | <graphical interface gate definition>

<graphical gate definition> ::=
{ <gate symbol> | <inherited gate symbol> }
is associated with { <gate> [<signal list area>] [<signal list area>] }***set***
[***is connected to*** <endpoint constraint>]

<endpoint constraint> ::=
{ <block symbol> | <process symbol> | <state symbol> }
contains <textual endpoint constraint>

<graphical interface gate definition> ::=
    <gate symbol 1>
    is associated with <interface identifier>

<gate symbol> ::=
    <gate symbol 1> | <gate symbol 2>

<gate symbol 1> ::=
    ————►

<gate symbol 2> ::=
    ◄————►

<inherited gate symbol> ::=
    <inherited gate symbol 1> | <inherited gate symbol 2>

<inherited gate symbol 1> ::=
    ---------►

<inherited gate symbol 2> ::=
    ◄---------►

The <gate on diagram> is outside the diagram frame.

A <graphical gate definition> that is part of a <gate property area> must not contain an <endpoint constraint>.

The <signal list area> elements are associated with the directions of the gate symbol.

<signal list area>s and <endpoint constraint> associated with an <inherited gate symbol> are regarded as additions to those of the gate definition in the supertype.

An <inherited 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

<textual interface gate definition> and <graphical interface gate definition> are shorthand for a <textual gate definition> or a <graphical gate definition>, respectively, having the name of the interface as <gate name> and the <interface identifier> as the <gate constraint> or <signal list area>.

8.2 Context parameters

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 context parameters as defined in 8.1.2.

The following type definitions can have formal context parameters: system type, block type, process type, procedure, signal, composite state, interface and data type.

Context parameters can be given constraints, that is, 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 list> <context parameters end>

<formal context parameter list> ::=
        <formal context parameter> {<end> <formal context parameter> }*

<actual context parameters> ::=
        <context parameters start> <actual context parameter list> <context parameters end>

<actual context parameter list> ::=
        [<actual context parameter>] {, [<actual context parameter> ] }*

<actual context parameter> ::=
        <identifier> | <constant primary>

<context parameters start> ::=
        <less than sign>

<context parameters end> ::=
        <greater than sign>

<formal context parameter> ::=
            <agent type context parameter>
        |   <agent 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>
        |   <exception context parameter>
        |   <composite state type context parameter>
        |   <gate context parameter>
        |   <interface context parameter>
```

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.

An <actual context parameter> shall not be a <constant primary> unless it is for a synonym context parameter. A <constant primary> is a <primary> that is a valid <constant expression> (see 12.2.1).

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 that 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 8.1.2.

A context parameter using other context parameters in its constraint cannot be bound before the other parameters are bound.

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> 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 (that is, deriving their qualifier, see 6.3), 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.

8.2.1 Agent type context parameter

Concrete textual grammar

<agent type context parameter> ::=
{**process type** | **block type**} <agent type name> [<agent type constraint>]

<agent type constraint> ::=
atleast <agent identifier> | <agent signature>

An actual agent type parameter must be a subtype of the constraint agent type (**atleast** <agent type identifier>) with no addition of formal parameters to those of the constraint type, or it must be compatible with the formal agent signature.

An agent type definition is compatible with the formal agent signature if it has the same kind and if the formal parameters of the agent type definition have the same sorts as the corresponding <sort>s of the <agent signature>.

8.2.2 Agent context parameter

Concrete textual grammar

<agent context parameter> ::=
{ **process** | **block** } <agent name> [<agent constraint>]

<agent constraint> ::=
{ **atleast** | <colon> } <agent type identifier> | <agent signature>

<agent signature> ::=
<sort list>

An actual agent parameter must identify an agent definition. Its type must be a subtype of the constraint agent type (**atleast** <agent type identifier>) with no addition of formal parameters to those of the constraint type, or it must be the type denoted by <agent type identifier>, or it must be compatible with the formal <agent signature>.

An agent definition is compatible with the formal <agent signature> if the formal parameters of the agent definition have the same sorts as the corresponding <sort>s of the <agent signature> or <agent formal parameters>, and both definitions have the same *Agent-kind*.

8.2.3 Procedure context parameter

Concrete textual grammar

```
<procedure context parameter> ::=
                    procedure <procedure name> <procedure constraint>

<procedure constraint> ::=
                    atleast <procedure identifier> | <procedure signature in constraint>

<procedure signature in constraint> ::=
                    [ (<formal parameter> { , <formal parameter> }* ) ] [<result>]
```

An actual procedure parameter must identify a procedure definition that is either a specialization of the procedure of the constraint (**atleast** <procedure identifier>) or is compatible with the formal procedure signature.

A procedure definition is compatible with the formal procedure signature if:

a) 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 have a result of the same <sort> or if neither returns a result; or

b) each **in**/**out** and **out** parameter in the procedure definition has the same <sort identifier> or <syntype identifier> as the corresponding parameter of the signature.

8.2.4 Remote procedure context parameter

Concrete textual grammar

```
<remote procedure context parameter> ::=
                    remote procedure <procedure name> <procedure signature in constraint>
```

An actual parameter to a **remote** procedure context parameter must identify a <remote procedure definition> with the same signature.

8.2.5 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>
```

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.

Semantics

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.

8.2.6 Variable context parameter

Concrete textual grammar

```
<variable context parameter> ::=
                    dcl <variable name> { , <variable name>}* <sort>
                            { , <variable name> { , <variable name>}* <sort> }*
```

An actual parameter must be a variable or a formal agent or procedure parameter of the same sort as the sort of the constraint.

8.2.7 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> }*
```

An actual parameter must identify a <remote variable definition> of the same sort.

8.2.8 Timer context parameter

Concrete textual grammar

```
<timer context parameter> ::=
                              timer <timer name> [<timer constraint>]
                                   { , <timer name> [<timer constraint>] }*

<timer constraint> ::=
                              <sort list>
```

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.

8.2.9 Synonym context parameter

Concrete textual grammar

```
<synonym context parameter> ::=
                              synonym <synonym name> <synonym constraint>
                                   {, <synonym name> <synonym constraint> }*

<synonym constraint> ::=
                              <sort>
```

An actual synonym must be a constant expression of the same sort as the sort of the constraint.

Model

If the actual parameter is a <constant expression> (rather than a <synonym identifier>), there is an implied definition of an anonymous synonym in the context surrounding the type being defined with the context parameter.

8.2.10 Sort context parameter

Concrete textual grammar

```
<sort context parameter> ::=
                              [{ value | object }] type <sort name> [<sort constraint>]

<sort constraint> ::=
                              atleast <sort> | <sort signature>

<sort signature> ::=
                              literals <literal signature> { , <literal signature> }*
                              [ operators <operation signature in constraint> { , <operation signature in constraint> }* ]
                              [ methods <operation signature in constraint> { , <operation signature in constraint> }* ]
                         |    operators <operation signature in constraint> { , <operation signature in constraint> }*
                              [ methods <operation signature in constraint> { , <operation signature in constraint> }* ]
                         |    methods <operation signature in constraint> { , <operation signature in constraint> }*

<operation signature in constraint> ::=
                              <operation name>[ (<formal parameter> { , <formal parameter> }* ) ] [<result>]
                         |    <name class operation> [<result>]
```

If <sort constraint> is omitted, the actual sort can be any sort. Otherwise, an actual sort must be either a subtype without <renaming> 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 operations defined by the data type that introduced the sort include the operations in the formal sort signature and the operations have the same signatures.

The <literal signature> must not contain <named number>.

Model

If the keyword **value** is given and the actual sort is an object sort, then the actual parameter is treated as the expanded sort **value** <sort identifier>. If the keyword **object** is given and the actual sort is a value sort, then the actual parameter is treated as the reference sort **object** <sort identifier>.

8.2.11 Exception context parameter

Concrete textual grammar

```
<exception context parameter>::=
                    exception <exception name> [<exception constraint>]
                         { , <exception name> [<exception constraint>] }*

<exception constraint>::=
                    <sort list>
```

An actual exception parameter must identify an exception with the same signature.

8.2.12 Composite state type context parameter

Concrete textual grammar

```
<composite state type context parameter> ::=
                    state type <composite state type name> [<composite state type constraint>]

<composite state type constraint> ::=
                    atleast <composite state type identifier> | <composite state type signature>

<composite state type signature> ::=
                    <sort list>
```

An actual composite state type parameter must identify a composite state type definition. Its type must be a subtype of the constraint composite state type (**atleast** <composite state type identifier>) with no addition of formal parameters to those of the constraint type or it must be compatible with the formal composite state type signature.

A composite state type definition is compatible with the formal composite state type signature if the formal parameters to the composite state type definition have the same sorts as the corresponding <sort>s of the <composite state type constraint>.

8.2.13 Gate context parameter

Concrete textual grammar

```
<gate context parameter> ::=
                    gate <gate> <gate constraint> [<gate constraint>]
```

An actual gate parameter must identify a gate definition. Its outward gate constraint must contain all elements mentioned in the <signal list> of the corresponding formal gate context parameter. The inward gate constraint of the formal gate context parameter must contain all elements in the <signal list> of the actual gate parameter.

8.2.14 Interface context parameter

Concrete textual grammar

```
<interface context parameter> ::=
                                        interface <interface name> [<interface constraint>]
                                                  { , <interface name> [<interface constraint>] }*

<interface constraint> ::=
                                        atleast <interface identifier>
```

An actual interface parameter must identify an interface definition. The type of the interface must be a subtype of the interface type of the constraint (**atleast** <interface identifier>).

8.3 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. Except in the case of interfaces, there is at most one supertype.

Virtual types can be given constraints, that is, properties any redefinition of the virtual type must have. These properties are used to guarantee properties of any redefinition. Virtual types are defined in 8.3.2.

8.3.1 Adding properties

Concrete textual grammar

```
<specialization> ::=
                                        inherits <type expression> [adding]
```

Concrete graphical grammar

```
<specialization area> ::=
                                        <specialization relation symbol>
                                        [ is associated with <actual context parameters> ]
                                        is connected to <type reference area>

<specialization relation symbol> ::=
                                        ——————————▷
```

The arrow end of the <specialization relation symbol> points towards the <type reference area>. The type connected to the arrow end is the supertype, while the other type is the subtype. The connected references must both be of the same kind. The associated binding of context parameters corresponds to the supertype being a type expression with actual context parameters.

<type expression> denotes the base type. The base type is said to be the supertype of the specialized type, and the specialized type is said to be a subtype of the base type. Any specialization of the subtype is also a subtype of the base type.

If a type subT is a subtype of a (super) type T (either directly or indirectly), then:

a) T must not enclose subT;

b) T must not be a specialization of subT;

c) definitions enclosed by T must not be specializations of subT.

In case of agent types, these rules must also hold for definitions enclosed in T and, in addition, definitions directly or indirectly enclosed by T must not be typebased definitions of subT.

The concrete syntax for specialization of data types is shown in 12.1.3.

Semantics

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 6.3. However, exceptions to this rule are:

a) a redefinition of a virtual type is a definition with the same name as that of the virtual type;

b) 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;

c) 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;

d) an operator of the supertype is not inherited if the signature of the specialized operator would be the same as the signature of the base type operator;

e) an operator or non-virtual method (that is, a method that is neither virtual nor redefined) of the supertype is not inherited if an operator or method with a signature equal to the signature of the specialized operator or method is already present in the subtype.

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 8.2).

The formal parameters of a specialized agent type are the formal parameters of the agent supertype followed by the formal parameters added in the specialization.

The formal parameters of a specialized procedure are the formal parameters of the procedure with the formal parameters added in the specialization. If the procedure before specialization has a <procedure result>, the parameters added in the specialization are inserted before the last parameter (the **out** parameter for the result), otherwise they are inserted after the last parameter.

The complete valid input signal set of a specialized agent type is the union of the complete valid input signal set of the specialized agent type and the complete valid input signal set of the agent supertype respectively.

The resulting graph of a specialized agent type, procedure definition or state type consists of the graph of its supertype definition followed by the graph of the specialized agent type, procedure definition or state type.

The state-transition graph of a given agent type, procedure definition or state type may have at most one unlabelled start transition.

A specialized signal definition may add (by appending) sorts to the sort list of the supertype.

A specialized data type definition may add literals, fields, or choices to the inherited type constructors, it may add operators and methods, and it may add default initializations or default assignment.

The formal parameters of a specialized composite state type are the formal parameters of the composite state type with the formal parameters added in the specialization.

NOTE – When a gate in a subtype is an extension of gate inherited from a supertype, the <inherited gate symbol> is used in the concrete graphical syntax.

8.3.2 Virtual type

An agent type, procedure or state type may be specified as a virtual type when it is defined locally to another type (denoted as the *enclosing* type). A virtual type may be redefined in specializations of the enclosing type.

Concrete textual grammar

```
<virtuality> ::=
                    virtual | redefined | finalized

<virtuality constraint> ::=
                    atleast <identifier>
```

<virtuality> and <virtuality constraint> are part of the type definition.

A virtual type is a type having **virtual** or **redefined** as <virtuality>. A redefined type is a type having **redefined** or **finalized** as <virtuality>. Only virtual types may be redefined.

Every virtual type has associated a virtuality constraint which is an <identifier> of the same entity kind as the virtual type. If <virtuality constraint> is specified, the virtuality constraint is the contained <identifier>; otherwise the virtuality constraint is derived as described below.

A virtual type and its constraints cannot have context parameters.

Only virtual types may have <virtuality constraint> specified.

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 agent type must have exactly the same formal parameters, and at least the same gates and interfaces with at least the definitions as those of its constraint. A virtual state type must have at least the same state connection points as its constraint. A virtual procedure must have exactly the same formal parameters as its constraint. The restrictions on the arguments of virtual operators and methods are given in 8.3.4.

If both **inherits** and **atleast** are used, then the inherited type must be identical to or be a subtype of the constraint.

In the case of an implicit constraint, redefinition involving **inherits** must be a subtype of the 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 (that is, a type name in the inheritance chain). The qualifier will consist of only one path item denoting the particular supertype.

A virtual or redefined type that has no <specialization> given explicitly may have an implicit <specialization>. The virtuality constraint and the possible implicit <specialization> are derived as below.

For a virtual type V and a redefined type R of V, then the following rules apply (all rules are applied in the given order):

a) if the virtual type V has no <virtuality constraint>, the constraint VC for type V is the same as the virtual type V and denotes the type V, otherwise the constraint VC is identified by the <virtuality constraint> given with type V;

b) if the virtual type V has no <specialization> and the constraint VC is the type V, type V does not have an implicit specialization;

c) if the virtual type V has no <specialization> and the constraint VC is not the type V, the implicit specialization type VS is the same as the constraint VC;

d) if <specialization> of the virtual type V is present, the specialization type VS must be the same as or a subtype of the constraint VC;

e) if the redefined type R has no <virtuality constraint>, the constraint RC for type R is the same as the type R, otherwise the constraint RC is identified by the <virtuality constraint> given with type R;

f) if the redefined type R has no <specialization>, the implicit specialization type RS for R is the same as the constraint VC from the type V, otherwise the specialization type RS is identified by the explicit <specialization> with type R;

g) the constraint RC must be the same as or a subtype of the constraint VC;

h) specialization type RS for R must be the same as or a subtype of the constraint RC;

i) if R is a virtual type (redefined rather than finalized), the same rules apply for R as for V.

A subtype of a virtual type is a subtype of the original virtual type and not of a possible redefinition.

8.3.3 Virtual transition/save

Transitions or saves of a process type, state 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 with the same state or signal, respectively, and with the keyword **redefined** or **finalized**.

Concrete textual grammar

The syntax of virtual transition and save are introduced in 9.4 (virtual procedure start), 10.5 (virtual remote procedure input and save), 11.1 (virtual process start), 11.2.2 (virtual input), 11.4 (virtual priority input), 11.5 (virtual continuous signal), 11.7 (virtual save), 11.9 (virtual spontaneous transition), and 11.16.3 (virtual handle).

Virtual transitions or saves must not appear in agent (set of instances) definitions, or in composite state 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 8.3.2).

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 handle transition can be redefined to a new virtual handle transition.

A virtual continuous transition can be redefined to a new continuous transition. The redefinition is indicated by the same state and priority (if present) 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 Annex F.

8.3.4 Virtual methods

Methods of a data type are specified to be virtual methods by means of the keyword **virtual** in <virtuality>. Virtual methods may be redefined in specializations. This is indicated by methods with the same <operation name> and with the keyword **redefined** or **finalized** in <virtuality>.

If the derived type contains only an <operation signature> but no <operation definition>, <textual operation reference>, or <external operation definition> for the redefined method, then only the signature of the redefined method is changed.

Concrete textual grammar

The syntax of virtual methods is introduced in 12.1.4.

When a method is redefined in a specialization, its signature must be sort compatible with the corresponding signature in the base type, and further, if the *Result* in the *Operation-signature* denotes a sort A, then the *Result* of the redefined method may only denote a sort B such that B is sort compatible to A.

A redefinition of a virtual method must not change the <parameter kind> in any <argument> of the inherited <operation signature>.

A redefinition of a virtual method must not add <argument virtuality> to any <argument> of the inherited <operation signature>.

Semantics

Virtual methods do not have a <virtuality constraint> which, in this case only, does not limit redefinition.

Redefinition of virtual methods corresponds closely to redefinition of virtual types (see 8.3.2).

8.3.5 Virtual default initialization

This subclause describes virtual default initialization, as introduced in 12.3.3.2.

Default initialization of instances of a data type is specified to be virtual by means of the keyword **virtual** in <virtuality>. A virtual default initialization may be redefined in specializations. This is indicated by a default initialization with the keyword **redefined** or **finalized** in <virtuality>.

If the derived type contains no <constant expression> in its default initialization, then the derived type does not have a default initialization.

Concrete textual grammar

The syntax of virtual default initializations was introduced in 12.3.3.2.

Semantics

Redefinition of a virtual default initialization corresponds closely to redefinition of virtual types (see 8.3.2).

8.4 Associations

An association expresses a binary relationship between two entity types, not necessarily distinct. Associations are intended to provide structured annotations to indicate additional properties of the types the associations are connected to, in a diagram or definition containing type references. The meaning of these properties is not defined by this Recommendation; that is, the meaning can be defined by some other Recommendation or standard or common specification or common understanding. An SDL system that contains an association has the same meaning and behaviour (as defined by this Recommendation), if the association is deleted.

Concrete textual grammar

```
<association> ::=
                        association [<association name>] <association kind> <end>

<association kind>::=
                        <association not bound kind>
                   |    <association end bound kind>
                   |    <association two ends bound kind>
                   |    <composition not bound kind>
                   |    <composition part end bound kind>
                   |    <composition composite end bound kind>
                   |    <composition two ends bound kind>
                   |    <aggregation not bound kind>
                   |    <aggregation part end bound kind>
                   |    <aggregation aggregate end bound kind>
                   |    <aggregation two ends bound kind>

<association not bound kind> ::=
                        from <association end> from <association end>

<association end bound kind>::=
                        from <association end> to <association end>

<association two ends bound kind>::=
                        to <association end> to <association end>

<composition not bound kind>::=
                        from <association end> composition <association end>

<composition part end bound kind>::=
                        to <association end> composition <association end>

<composition composite end bound kind> ::=
                        from <association end> to composition <association end>

<composition two ends bound kind>::=
                        to <association end> to composition <association end>

<aggregation not bound kind>::=
                        from <association end> aggregation <association end>

<aggregation part end bound kind>::=
                        to <association end> aggregation <association end>

<aggregation aggregate end bound kind>::=
                        from <association end> to aggregation <association end>

<aggregation two ends bound kind>::=
                        to <association end> to aggregation <association end>

<association end> ::=
                        [<visibility>] [as <role name>] <linked type identifier> [size <multiplicity>] [ordered]
```

<linked type identifier> ::=
        <agent type identifier>
    | <data type identifier>
    | <interface identifier>

<multiplicity> ::=
        <range condition>

An <association> is allowed to link agent types, interfaces or data types.

If an <association end> identifies an agent type or an interface, the **protected** visibility cannot be used in the other <association end> of the <association>.

If two different <association>s identify the same type, in the <association end>s opposite to this common type the <role name>s (if given) must be different.

The base sort of the <range condition> in <multiplicity> must be the Predefined Natural sort.

There must not be a set of <association>s containing composition such that a type is linked by composition back to itself, either directly or indirectly.

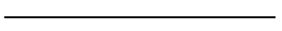
If an <association end> is preceded by the keyword **composition** and identifies a data type or interface, then the <linked type identifier> in the other <association end> of the <association> must identify a data type or interface, respectively.

Concrete graphical grammar

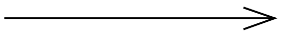
<association area> ::=
        <association symbol>
        [***is associated with*** <association name>]
        is connected to {<association end area> <association end area>} ***set***

<association symbol> ::=
        <association not bound symbol>
    | <association end bound symbol>
    | <association two ends bound symbol>
    | <composition not bound symbol>
    | <composition part end bound symbol>
    | <composition composite end bound symbol>
    | <composition two ends bound symbol>
    | <aggregation not bound symbol>
    | <aggregation part end bound symbol>
    | <aggregation aggregate end bound symbol>
    | <aggregation two ends bound symbol>

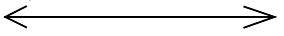
<association not bound symbol> ::=



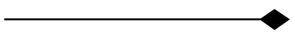
<association end bound symbol>::=



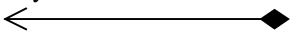
<association two ends bound symbol>::=



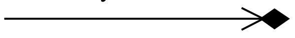
<composition not bound symbol> ::=



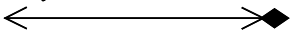
<composition part end bound symbol> ::=



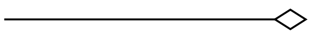
<composition composite end bound symbol> ::=



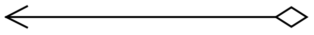
<composition two ends bound symbol> ::=



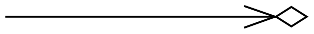
<aggregation not bound symbol> ::=



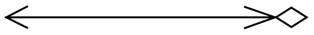
<aggregation part end bound symbol> ::=



<aggregation aggregate end bound symbol> ::=



<aggregation two ends bound symbol> ::=



<association end area> ::=
  <linked type reference area> ***is associated with***
    { [<<u>role</u> name>] [<multiplicity>] [<ordering area>] [<symbolic visibility>] }***set***

<ordering area> ::=
  ordered

<linked type reference area> ::=
  <agent type reference area>
| <data type reference area>
| <interface reference area>

An <association area> connecting two <linked type reference area>s corresponds to an <association> in the enclosing diagram.

If an <association end area> identifies an agent type or an interface, **protected** visibility shall not be used in the other <association end area> of the <association area>.

If two different <association area>s identify the same type, in the <association end area>s opposite to this common type the <<u>role</u> name>s (if given) must be different.

There must not be a set of <association area>s containing composition such that a type is linked by composition back to itself, either directly or indirectly.

If the composite end (the end with a diamond) of a <composition not bound symbol>, <composition part end bound symbol>, <composition composite end bound symbol>, or <composition two ends bound symbol> is connected to a <linked type reference area> that identifies a data type or interface, the opposite <association end area> must be connected to a <linked type reference area> that identifies a data type or interface, respectively.

Semantics

An association links the two entity types in some way not further defined by this Recommendation.

9 Agents

An agent definition defines an (arbitrarily large) set of agents. An agent is characterized by having variables, procedures, a state machine (given by an explicit or implicit composite state type) and sets of contained agents.

There are two kinds of agents: *blocks* and *processes*. A *system* is the outermost block. The state machine of a block is interpreted *concurrently* with its contained agents, while the state machine of a process is interpreted *alternating* with its contained agents.

Abstract grammar

Agent-definition	::	*Agent-name*
		Number-of-instances
		Agent-type-identifier
State-machine-definition	::	*State-name*
		Composite-state-type-identifier
Agent-kind	=	**SYSTEM | BLOCK | PROCESS**
Number-of-instances	::	*Initial-number* [*Maximum-number*]
Initial-number	=	*Nat*
Maximum-number	=	*Nat*

```
State-transition-graph          ::    [On-exception]
                                      State-start-node
                                      State-node-set
                                      Free-action-set
                                      Exception-handler-node-set
Agent-formal-parameter          =     Parameter
```

Concrete textual grammar

```
<agent definition> ::=
                    <system definition> | <block definition> | <process definition>

<agent structure> ::=
                    [<valid input signal set>]
                    {      {      <entity in agent>
                           |      <channel to channel connection>
                           |      <channel definition>
                           |      <gate in definition>
                           |      <agent definition>
                           |      <agent reference>
                           |      <textual typebased agent definition> }*
                           [ <state partitioning>]
                    |      {      <entity in agent>
                           |      <gate in definition> }*
                           <agent body> }
```

The <state partitioning> must have the same name as the containing agent. It defines the state machine of the agent. If <agent structure> can be understood both as <state partitioning> and <agent body>, it is interpreted as <agent body>.

```
<agent instantiation> ::=
                    [<number of instances>]
                    <agent additional heading>

<agent additional heading> ::=
                    [<specialization>] [<agent formal parameters>]

<entity in agent> ::=
                    <signal definition>
               |    <signal reference>
               |    <signal list definition>
               |    <variable definition>
               |    <remote procedure definition>
               |    <remote variable definition>
               |    <data definition>
               |    <data type reference>
               |    <timer definition>
               |    <interface reference>
               |    <macro definition>
               |    <exception definition>
               |    <procedure reference>
               |    <procedure definition>
               |    <composite state type definition>
               |    <composite state type reference>
               |    <select definition>
               |    <agent type definition>
               |    <agent type reference>
               |    <association>

<agent reference> ::=
                    <block reference>
               |    <process reference>

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

```
<agent body> ::=
                              <state machine graph>

<state machine graph>::=
                              [<on exception>] <start>
                              { <state> | <exception handler> | <free action> } *

<agent formal parameters> ::=
                              ( <parameters of sort> {, <parameters of sort>}* )

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

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

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

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

The following is valid for agents in general. Special properties of systems, blocks and processes are treated in separate clauses on these concepts.

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

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

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

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

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

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

Similar relations apply for <number of instances> specified in <agent diagram> and in <agent reference area> 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.

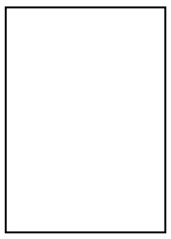
In <agent instantiation>, if <agent formal parameters> are present, <number of instances> must be present, even if both <initial number> and <maximum number> are omitted.

The use and syntax of <valid input signal set> is defined in clause 9.

Concrete graphical grammar

```
<agent diagram> ::=
                              <system diagram> | <block diagram> | <process diagram>

<agent diagram content> ::=
                              {      {<agent text area>}*
                                     {<type in agent area>}*
                                     { <interaction area> | <agent body area> } }set
```

```
<agent body area> ::=
                              <state machine graph area>

<frame symbol> ::=
```



The <package use area> must be placed on the top of the <frame symbol> of the <system diagram>, <block diagram>, or <process diagram>.

```
<agent text area> ::=
                              <text symbol>
                              contains {
                                     [<valid input signal set>]
                                     {      <signal definition>
                                     |      <signal reference>
                                     |      <signal list definition>
                                     |      <variable definition>
                                     |      <remote procedure definition>
                                     |      <remote variable definition>
                                     |      <data definition>
                                     |      <data type reference>
                                     |      <timer definition>
                                     |      <interface reference>
                                     |      <macro definition>
                                     |      <exception definition>
                                     |      <procedure definition>
                                     |      <procedure reference>
                                     |      <select definition>
                                     |      <agent type reference>
                                     |      <agent reference> }* }

<state machine graph area>::=
                              {      [ <on exception association area> ] <start area>
                                     { <state area> | <exception handler area> | <in connector area> }* }set

<interaction area> ::=
                              {{       <agent area>
                                     | <create line area>
                                     | <channel definition area>
                                     | <state machine area> }+ }set

<agent area> ::=
                              <agent reference area>
                        |     <agent diagram>
                        |     <graphical typebased agent definition>
                        |     <inherited agent definition>

<agent reference area> ::=
                              {      <system reference area>
                              |      <block reference area>
                              |      <process reference area> }
                              [ is connected to { <graphical package use area>+ }set ]

<state machine area> ::=
                              <state partition area>

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

```
<create line endpoint area> ::=
                                <agent area> | <agent type area> | <state machine area>

<agent type area> ::=
                                <agent type reference area>
                        |       <agent type diagram>

<create line symbol> ::=
                                <dependency symbol>
```

An <agent text area> is permitted to contain an <agent reference> only if the directly enclosing <agent diagram content> contains an <interaction area>.

In an <agent diagram> the <gate on diagram>s must be outside the diagram frame.

The *Agent-definition*-**set** in the *Abstract grammar* of the implied agent type (see *Model*) corresponds to the <agent area>s.

The *Channel-definition*-**set** in the *Abstract grammar* of the implied agent type corresponds to the <channel definition area>s.

The arrowhead on the <create line symbol> indicates the <agent area> or <agent type area> of an agent or agent type upon which a create action is performed. <create line symbol>s are optional, but if used then there must be a create request for the agent (or agent type) at the arrowhead end of the <create line symbol> in the agent (or agent type or state machine) at the originating end of the <create line symbol>. The create action can be inherited and need thus not be specified directly in the agent or agent type. This rule applies after transformation of <option area>.

NOTE 1 – This rule can be independently applied before or after transformation of <transition option>.

The <state machine area> of <interaction area> identifies the state machine (composite state) of the agent, which may be given directly as an agent graph or by reference to a state definition.

If there is an <agent reference area> for the agent, there must be a <gate property area> connected to the <agent reference area> or there must be a <gate> contained in the symbol for each <gate on diagram> connected to the <agent diagram>. If the <gate on diagram> is a <graphical gate definition> then the <gate property area> must be a <graphical gate definition> or <gate>, respectively, mentioning the same <gate name>. If the <gate on diagram> is a <graphical interface gate definition> then the <gate property area> must be a <graphical interface gate definition> mentioning the same <interface identifier>. A corresponding rule applies if there is an <agent reference> for the agent.

A <graphical package use area> connected to a <agent reference area> must be consistent with the <graphical package use area> of the referenced diagram.

Semantics

An *Agent-definition* has a name which can be used in qualifiers in conjunction with **system**, **block** or **process** depending on the kind of the agent.

An agent definition defines a set of agents. Several instances of the same agent set may exist at the same time and be interpreted asynchronously and in parallel or alternating with each other and with instances of other agent sets in the system.

The first value in the *Number-of-instances* represents the number of instances of the agent set which exist when the system or containing entity is created (initial instances), the second value represents the maximum number of simultaneous instances of the agent set.

The behaviour of an *Agent-definition* in an *Agent-definition-**set*** depends on whether the containing *Agent-definition* is a block or process, and therefore is defined for block and process separately.

An agent instance has a communicating extended finite state machine defined by its explicit or implicit state machine definition. Whenever the state machine is in a state, on input of a given signal it will perform a certain sequence of actions, denoted as a transition. The completion of the transition results in the state machine of the agent instance waiting in another state, which is not necessarily different from the first one.

When an agent is interpreted, the initial agents it contains are created. The signal communication between the finite state machines of these initial agents, the finite state machine of the agent and their environment commences only when all the initial agents have been created. The time taken to create an agent may or may not be significant. The formal parameters of the initial agents have no associated data items (they are "undefined").

Agent instances exist from the time that the containing agent is created or they can be created by create request actions of agents being interpreted; their interpretations start when their start action is interpreted; they may cease to exist by performing stop actions.

When the state machine of an agent interprets a stop, if this agent was a concurrent container it will continue to handle the implicit remote procedure calls mediating the access to the global variables. The state machine of such an agent remains in this “stopping condition” until all contained agents have terminated, after which the agent terminates. While in the stopping condition, the agent will not accept any stimuli other than the implicit set and get remote procedure calls introduced for each global variable, if any. After an agent has terminated, its pid is no longer valid.

Signals received by agent instances are denoted as input signals, and signals sent from agent instances are denoted as output signals. <valid input signal set> of an agent defines the <valid input signal set> of its state machine.

Calling and serving remote procedure calls, and accessing remote variables, also correspond to exchange of signals (see 10.5 and 10.6 respectively).

Signals may be consumed by the state machine of an agent instance only when it is in a state. The complete valid input signal set is the union of:

a) the set of signals in all channels or gates leading to the set of agent instances;

b) the <valid input signal set>;

c) the implicit input signals introduced as in 10.5, 10.6, 11.4, 11.5 and 11.6; and

d) the timer signals.

Exactly one input port is associated with the finite state machine of each agent instance. Signals that are sent to a container agent are delivered to the input port of the agent, provided that the signal appears on a (explicit or implicit) channel connected to its state machine. Signals occurring only in the <valid input signal set> must not be used for external communication. They serve for the communication between instances within the same instance set.

The finite state machine of an agent is either waiting in a state or active, performing a transition. For each state, there is a save signal set (see also 11.7). 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 agent. 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 can be queued for the finite state machine of the agent instance. The set of retained signals is 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 agent is created, its finite state machine is given an empty input port, and local variables of the agent are created.

When a container agent instance is created, the initial agents of the contained agent sets are created. If the container is created by a <create request>, **parent** of the contained agents (see *Model* below) receives the pid of the container. 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 agent instance is dynamically created.

If an <agent definition> or an <agent type definition>, which is used in a <textual typebased agent definition>, contains <channel definition>s and <textual typebased agent definition>s, then each gate of the <agent type definition>s of the contained <textual typebased agent definition>s must be connected to a channel.

The definition of an agent implies the definition of an interface in the same scope of the agent (see 12.1.2). The pid sort implicitly defined by this interface is identified with *Agent-name* and is visible in the same scope unit as where the agent is defined.

NOTE 2 – Because every agent has an implicitly defined interface with the same name, the agent must have a different name from every explicitly defined interface, and every agent type (these also have implicit interfaces) defined in the same scope, otherwise there are names clashes.

Model

An *Agent-definition* has an implied anonymous agent type that defines the properties of the agent.

An agent with an <agent body> or an <agent body area> is shorthand for an agent having only a state machine, but no contained agents. This state machine is obtained by replacing the <agent body> or <agent body area> by a composite state definition. This composite state definition has the same name as the agent and its *State-transition-graph* is represented by the <agent body> or the <agent body area>.

An agent that is a specialization is shorthand for defining an implicit agent type and one typebased agent of this type.

In all agent instances, four anonymous variables of the pid sort of the agent (for agents not based on an agent type) or the pid sort of the agent type (for type based agents) are declared and are, in the following, referred to by **self**, **parent**, **offspring** and **sender**. They give a result for:

a) the agent instance (**self**);

b) the creating agent instance (**parent**);

c) the most recent agent instance created by the agent instance (**offspring**);

d) the agent instance from which the last input signal has been consumed (**sender**) (see also 11.2.2).

These anonymous variables are accessed using pid expressions as further explained in 12.3.4.3.

For all agent instances present at system initialization, **parent** is initialized to Null.

For all newly created agent instances, **sender** and **offspring** are initialized to Null.

9.1 System

A System is the outermost agent and has the *Agent-kind* **SYSTEM**. It is defined by a <system definition> or a <system diagram>. The semantics of agents applies with the additions provided in this subclause.

Abstract grammar

An *Agent* with the *Agent-kind* **SYSTEM** must not be contained in any other *Agent*. It must contain either at least one *Agent-definition* or an explicit or implicit *State-machine-definition*.

The definitions of all signals, channels, data types and syntypes, used in the interface with the environment and between contained agents of the system (including itself) are contained in the *Agent-definition* of the system.

The *Initial-number* of instances is 1 and the *Maximum-number* of instances is 1.

NOTE – <number of instances> cannot be specified.

Concrete textual grammar

```
<system definition> ::=
                        {<package use clause>}*
                        <system heading> <end> <agent structure>
                        endsystem [<system name>] <end>

<system heading>::=
                        system <system name> <agent additional heading>
```

The <agent additional heading> in a <system definition> or a <system diagram> may not include <agent formal parameters>.

Concrete graphical grammar

```
<system diagram> ::=
                        <frame symbol> contains {<system heading> <agent diagram content> }
                        [ is associated with <package use area> ]

<system reference area> ::=
                        <block symbol> contains
                              { system <system name> }
```

The <gate on diagram>s in a <system diagram> may not include <channel identifier>s.

A <system reference area> must only be used as part of an <sdl specification>.

Semantics

An *Agent-definition* with the *Agent-kind* **SYSTEM** is the SDL representation of a specification or description of a system. A system is the outermost block. This means that agents within a system are blocks and processes that are interpreted concurrently with each other and with the possible state machine of the system.

A system is separated from its environment by a system boundary and contains a set of agents. Communication between the system and the environment or between agents within the system can take place using signals, remote procedures and remote variables. Within a system, these communication means are conveyed on explicit or implicit channels. The channels connect the contained agents to one another or to the system boundary.

A system instance is an instantiation of a system type defined by an *Agent-definition* with the *Agent-kind* **SYSTEM**. The interpretation of a system instance is performed by an abstract SDL machine, which thereby gives semantics to the SDL concepts. To interpret a system instance is to:

a) initiate the system time;

b) interpret the contained agents and their connected channels; and

c) interpret the optional state machine of the system.

Model

For each <channel definition> in a system mentioning **env**, a gate with an anonymous name is added to the Agent-definition. The channel definition is changed to mention this gate in the <channel path> directed to the system environment.

9.2 Block

A block is an agent with the *Agent-kind* **BLOCK**. The semantics of agents therefore applies with the additions provided in this subclause. A block is defined by a <block definition> or a <block diagram>.

The instances contained within a block instance are interpreted concurrently and asynchronously with each other and with the state machine of the containing block instance. All communication between different contained instances within a block is performed asynchronously using signal exchange, either explicitly or implicitly using for example remote procedure calls.

Concrete textual grammar

<block definition> ::=
    {<package use clause>}*
    <block heading> <end> <agent structure>
    endblock [[<qualifier>] <block name>] <end>

<block heading> ::=
    block [<qualifier>] <block name> <agent instantiation>

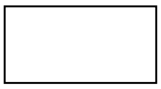
<block reference> ::=
    block <block name> [<number of instances>] **referenced** <end>

Concrete graphical grammar

<block diagram> ::=
    <frame symbol> ***contains*** {<block heading> <agent diagram content> }
    is connected to { {<gate on diagram>}* <external channel identifiers> }***set***
    [***is associated with*** <package use area>]

<block reference area> ::=
    <block symbol> ***contains***
        { { <block name> [<number of instances>] } {<gate>*}***set*** }
    is connected to { <gate property area>* }***set***

<block symbol> ::=



The <gate>s are placed near the border of the symbols and associated with the connection point to channels.

The <external channel identifiers> identify external channels connected to channels in the <block diagram>. It is placed outside the <frame symbol>, close to the endpoint of the channels at the <frame symbol>.

Semantics

A block definition is an agent definition that defines containers for one or more process or block definitions.

A block instance is an instantiation of a block type defined by an *Agent-definition* with the *Agent-kind* **BLOCK**. To interpret a block instance is to:

a) interpret the contained agents and their connected channels;

b) interpret the state machine of the block (if present).

In a block with a finite state machine, the finite state machine is created as part of the creation of the block (and its contained agents), and it is interpreted concurrently with the agents in the block.

A block with a variable definition but no state machine has an associated implicit state machine that is interpreted concurrently with agents in the block.

Access from contained agents in the block to a variable of the block is covered by two implicitly defined remote procedures for setting and getting the data item associated with the variable. These procedures are provided by the state machine of the block.

Model

A block b with a state machine and variables is modelled by keeping the block b (without the variables) and transforming the state entity and variables into a separate state machine (sm) in the block b. For each variable v in b, this state machine will have a variable v and two exported procedures set_v (with an IN parameter of the sort of v) and get_v (with a return type being the sort of v). Each assignment to v from enclosed definitions is transformed to a remote call of set_v. Each occurrence of v in expressions in enclosed definitions is transformed to a remote call of get_v. These occurrences also apply to occurrences in procedures defined in block b, as these are transformed into procedures local to the calling agents.

A block b with only variables and/or procedures is transformed as above, with the graph of the generated state machine having just one state, where it inputs the generated set and get procedures.

The channels connected to the state machine are transformed so that they are connected to sm.

This transformation takes place after types and context parameters have been transformed.

9.3 Process

A process is an agent with the *Agent-kind* **PROCESS**. The semantics of agents therefore applies with the additions provided in this subclause. A process is defined by a <process definition> or a <process diagram>.

A process is used to introduce shared data into a specification, allowing to use the variables of the containing process or by using objects. All instances in a process can access the local variables of the process.

To achieve safe communication despite the sharing of data in a process all instances are interpreted using alternating semantics. This implies that for any two instances inside a process no two transitions are interpreted in parallel and also that the interpretation of a transition in one instance is not interrupted by another instance. When an instance is waiting for example for a remote procedure call return, it is in a state; therefore an alternate instance can be interpreted.

Abstract grammar

An *Agent-definition* with the *Agent-kind* **PROCESS** must either contain at least one *Agent-definition* or it shall have an explicit or implicit *State-machine-definition*.

The contained *Agent-definitions* of an *Agent-definition* with the *Agent-kind* **PROCESS** shall all have the *Agent-kind* **PROCESS**.

Concrete textual grammar

<process definition> ::=
    {<package use clause>}*
    <process heading> <end> <agent structure>
    endprocess [[<qualifier>] <process name>] <end>

<process heading> ::=
    process [<qualifier>] <process name> <agent instantiation>

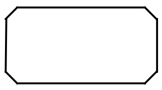
<process reference> ::=
    process <process name> [<number of instances>] **referenced** <end>

Concrete graphical grammar

<process diagram> ::=
    <frame symbol> ***contains*** {<process heading> <agent diagram content> }
    is connected to { {<gate on diagram>}* <external channel identifiers> }***set***
    [***is associated with*** <package use area>]

<process reference area> ::=
<process symbol> ***contains***
{ { <process name> [<number of instances>] } {<gate>*}***set*** }
is connected to { {<gate property area>*}***set*** }

<process symbol> ::=



The <gate>s are placed near the border of the symbols and associated with the connection point to channels.

The <external channel identifiers> identify external channels connected to channels in the <process diagram>. It is placed outside the <frame symbol>, close to the endpoint of the channels at the <frame symbol>.

Semantics

An instance of a process with contained process instance sets is interpreted by interpreting the instances in the contained process instance sets alternating with each other and with the state machine of the containing process instance, if any. Alternating interpretation implies that only one of the instances inside the alternating context can interpret a transition at a time, and also that once interpretation of a transition of an involved process instance has started, it continues until a (explicit or implicit) state is reached or the process instance terminates. The state can be an implicit state introduced by transformations (for example, due to a remote procedure call).

A process with variable definitions and contained processes, but without a state machine, has an associated implicit state machine that is interpreted alternating with the contained processes.

NOTE – State aggregation has also alternating interpretation. However, alternating processes of a process each have their own input port and their own **self**, **parent**, **offspring** and **sender**. In the case of state aggregation there is only one input port and one set of **self**, **parent**, **offspring** and **sender** belonging to the container agent.

9.4 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 expressions (value returning procedures only).

Abstract grammar

Procedure-definition	::	*Procedure-name*
		*Procedure-formal-parameter**
		[*Result*]
		Procedure-identifier
		*Data-type-definition-**set***
		*Syntype-definition-**set***
		*Variable-definition-**set***
		*Composite-state-type-definition-**set***
		*Procedure-definition-**set***
		Procedure-graph
Procedure-name	=	*Name*
Procedure-formal-parameter	=	*In-parameter*
	|	*Inout-parameter*
	|	*Out-parameter*
In-parameter	::	*Parameter*
Inout-parameter	::	*Parameter*
Out-parameter	::	*Parameter*

Parameter	::	*Variable-name*
		Sort-reference-identifier
Result	::	*Sort-reference-identifier*
Procedure-graph	::	[*On-exception*]
		Procedure-start-node
		*State-node-**set***
		*Free-action-**set***
		*Exception-handler-node-**set***
Procedure-start-node	::	[*On-exception*]
		Transition
Procedure-identifier	=	*Identifier*

If a *Procedure-definition* contains *Result*, it corresponds to a value returning procedure.

Concrete textual grammar

<procedure definition> ::=
    <external procedure definition>
  | {<package use clause>}*
    <procedure heading> <end>
        <entity in procedure>*
        [<procedure body>]
    endprocedure [[<qualifier>] <procedure name>] <end>
  | {<package use clause>}*
    <procedure heading>
        [<end> <entity in procedure>+]
        [<virtuality>] <left curly bracket>
            <statement list>
        <right curly bracket>

<procedure preamble> ::=
    <type preamble> [**exported** [**as** <remote procedure identifier>]]

<procedure heading> ::=
    <procedure preamble>
    procedure [<qualifier>] <procedure name>
    [<formal context parameters>] [<virtuality constraint>]
    [<specialization>]
    [<procedure formal parameters>]
    [<procedure result>] [<raises>]

<procedure formal parameters> ::=
    (<formal variable parameters> {**,** <formal variable parameters> }* **)**

<formal variable parameters> ::=
    <parameter kind> <parameters of sort>

<parameter kind> ::=
    [**in/out** | **in** | **out**]

<procedure result> ::=
    <result sign> [<variable name>] <sort>

<raises> ::=
    raise <exception identifier> {**,** <exception identifier>}*

<entity in procedure> ::=
    <variable definition>
  | <data definition>
  | <data type reference>
  | <procedure reference>
  | <procedure definition>
  | <composite state type definition>
  | <composite state type reference>
  | <exception definition>
  | <select definition>
  | <macro definition>

<procedure signature> ::=

[**(**<formal parameter> { **,** <formal parameter> }* **)**] [<result>] [<raises>]

<procedure body> ::=

[<on exception>] [<start>] { <state> | <exception handler> | <free action> }*

<external procedure definition> ::=

procedure <procedure name> <procedure signature> **external** <end>

<procedure reference> ::=

<type preamble>
procedure <procedure identifier> <type reference properties>

The optional <virtuality> before <left curly bracket> <statement list> in <procedure definition> applies to the start transition of the procedure, which in this case is the statement list.

An exported procedure cannot have formal context parameters and its enclosing scope must be an agent type or agent definition.

<variable definition> in a <procedure definition> cannot contain **exported** <variable name>s (see 12.3.1).

If present, **exported** is inherited by any subtype of a procedure. A virtual exported procedure must contain **exported** in all redefinitions. Virtual types including virtual procedures are described in 8.3.2. 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 an agent cannot mention the same <remote procedure identifier>.

An external procedure cannot be mentioned in a <type expression>, in a <formal context parameter>, or in a <procedure constraint>.

There is no corresponding graphical syntax for <external procedure definition>.

If an exception can be raised in a procedure when no exception handler is active with the corresponding handler clause (that is, it is not handled), the <raises> must mention this exception. An exception is considered as not handled in a procedure if there is a potential control flow inside the procedure producing that exception, and none of the exception handlers activated in this control flow handle the exception.

If a <procedure definition> appears inside of a <text symbol>, it must not contain <state>.

An **endprocedure** within an inner <procedure definition> in an <entity in procedure> of an outer <procedure definition> belongs to the inner <procedure definition>.

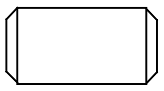
Concrete graphical grammar

```
<procedure diagram> ::=
                    <frame symbol> contains {
                          <procedure heading>
                          {     <procedure text area>*
                                <procedure area>*
                                <procedure graph area> }set }
                    [ is associated with <package use area> ]

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

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

<procedure reference area> ::=
                        <type reference area>
```

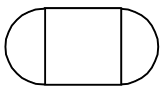
The <type reference area> that is part of a <procedure reference area> must have a <graphical type reference heading> that contains a <procedure name>.

<procedure symbol> ::=



```
<procedure graph area> ::=
                        [ <on exception association area> ] [<procedure start area>]
                            {<state area> | <exception handler area> | <in connector area> }*

<procedure start area> ::=
                        <procedure start symbol>
                        contains { [<virtuality>] }
                        [ is connected to <on exception association area> ]
                        is followed by <transition area>
```

<procedure start symbol> ::=



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

The <on exception association area> of a <procedure graph area> identifies the exception handler associated to the whole graph. The originating end must not be connected to any symbol.

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 10.5.

A procedure variable is a local variable within the procedure that cannot be exported. 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 *Call-node* (represented by a <procedure call>, see 11.13.3), a *Value-returning-call-node* (represented by a <value returning procedure call>, see 12.3.5), or an *Operation-application* (represented by an <operation application>, see 12.2.7) causes the creation of a procedure instance and the interpretation to commence in the following way:

a) A local variable is created for each *In-parameter*, having the *Name* and *Sort* of the *In-parameter*. The variable is associated with the result of the expression by interpreting an assignment between the variable and the expression given by the corresponding actual parameter if present. Otherwise the variable gets no associated data item; that is, it becomes "undefined".

b) A local variable is created for each *Out-parameter*, having the *Name* and *Sort* of the *Out-parameter*. The variable gets no data item; that is, it becomes "undefined".

c) A local variable is created for each *Variable-definition* in the *Procedure-definition*.

d) Each *Inout-parameter* denotes a variable that is given by the actual parameter expression in 11.13.3. The contained *Variable-name* is used throughout the interpretation of the *Procedure-graph* when referring to the data item associated to the variable or when assigning a new data item to the variable.

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

f) Before interpretation of a *Return-node* contained in the *Procedure-graph*, the *Out-parameters* are given the data items of the corresponding local variable.

The nodes of the procedure graph are interpreted in the same manner as the equivalent nodes of an agent; that is, the procedure has the same complete valid input signal set as the enclosing agent, and the same input port as the instance of the enclosing agent that has called it, either directly or indirectly.

An external procedure is a procedure whose <procedure body> is not included in the SDL description. Conceptually, an external procedure is assumed to be given a <procedure body> and will be transformed into a <procedure definition> as part of the generic system transformation (see Annex F).

Model

A formal parameter with no explicit <parameter kind> has the implicit <parameter kind> **in**.

When a <variable name> is present in <procedure result>, then all <return>s or <return area>s within the procedure graph without an <expression> are replaced by a <return> or <return area>, respectively, containing <variable name> as the <expression>.

A <procedure result> with <variable name> is derived syntax for a <variable definition> with <variable name> and <sort> in <variables of sort>. If there is a <variable definition> involving <variable name> no further <variable definition> is added.

A <procedure start area> which contains <virtuality>, a procedure <start> which contains <virtuality>, or a <statement list> in a <procedure definition> following <virtuality> is called a virtual procedure start. Virtual procedure start is further described in 8.3.3.

The second form of <procedure definition> is derived syntax for the following <procedure definition>:

```
<procedure preamble>
<procedure heading>;
   start <virtuality>;
   task { <statement list>-transform };
   return ;
endprocedure ;
```

This transformation takes place after the transformation of <compound statement>.

10 Communication

10.1 Channel

Abstract grammar

Channel-definition	::	*Channel-name*
		[**NODELAY**]
		*Channel-path**-set***
Channel-path	::	*Originating-gate*
		Destination-gate
		*Signal-identifier**-set***
Originating-gate	=	*Gate-identifier*
Destination-gate	=	*Gate-identifier*
Gate-identifier	=	*Identifier*
Agent-identifier	=	*Identifier*
Channel-name	=	*Name*

The *Channel-path**-set*** contains at least one *Channel-path* and no more than two. When there are two paths, the channel is bidirectional and the *Originating-gate* of each *Channel-path* must be the same as the *Destination-gate* of the other *Channel-path*.

If the *Originating-gate* and the *Destination-gate* are the same agent, the channel must be unidirectional (there must be only one element in the *Channel-path**-set***).

The *Originating-gate* or *Destination-gate* must be defined in the same scope unit in the abstract syntax in which the channel is defined.

NODELAY denotes that the channel has no delay.

A channel is allowed to connect the two directions of a bidirectional gate to each other.

Concrete textual grammar

```
<channel definition> ::=
                    channel [<channel name>] [nodelay]
                         <channel path> [<channel path>]
                    endchannel [<channel name>] <end>

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

<channel endpoint> ::=
                    { <agent identifier> | <state identifier> | env | this }[<via gate>]

<via gate> ::=
                    via <gate>
```

The ending <channel name> may only be specified if the starting <channel name> is specified. If the starting <channel name> is not specified, the channel cannot be referred to by name.

The <channel endpoint> **this** denotes the state machine of the agent directly enclosing the channel definition.

<gate> must be specified if:

a) <channel endpoint> denotes a connection to a <textual typebased agent definition> or a <textual typebased state partition definition> in which case the <gate> must be defined directly in the agent type or state type for that agent or state respectively; or

b) **env** is specified and the channel is defined in an agent type in which case the <gate> must be defined in this agent type respectively.

If <gate> is specified, the channel is connected to that gate. The gate and the channel must have at least one common element in their signal lists in the same direction. If no <gate> is specified, the following rules apply:

a) if the channel endpoint is an agent or state machine and that agent/state contains a <channel to channel connection> for the channel, the channel is connected to the implicit gate introduced by the <channel to channel connection>;

b) if the channel endpoint is a state, the channel is connected to the implicit gate of that state machine (see Annex F),

otherwise the channel introduces an implicit gate on the agent or state mentioned in <channel endpoint>. This gate obtains the <signal list> of the respective <channel path>s as its corresponding gate constraint. The channel is connected to that gate.

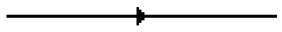
Concrete graphical grammar

<channel definition area> ::=
        <channel symbol>
        is associated with
            { [<channel name>] { [<gate>] [<signal list area>] [<signal list area>] }***set*** }
        is connected to {
            {   <agent area> | <state machine area>
            | <gate property area> | <gate on diagram> }
            {   <agent area> | <state machine area>
            | <gate property area> | <gate on diagram> } }***set***

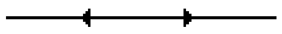
When a <channel symbol> is connected to a <state machine area>, it denotes the state machine of the agent directly enclosing the channel definition.

<channel symbol> ::=
          <delaying channel symbol 1>
    | <delaying channel symbol 2>
    | <nondelaying channel symbol 1>
    | <nondelaying channel symbol 2>

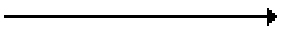
<delaying channel symbol 1> ::=



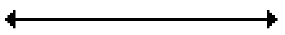
<delaying channel symbol 2> ::=



<nondelaying channel symbol 1> ::=



<nondelaying channel symbol 2> ::=



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

The arrowheads for <nondelaying channel symbol 1> and <nondelaying channel symbol 2> are placed at the end(s) of the channel and indicate that the channel has no delay.

Semantics

A *Channel-definition* represents a transportation path for signals (including the implicit signals implied by remote procedures and remote variables, see 10.5 and 10.6). A channel can be considered as one or two independent unidirectional channel paths between two agents or between an agent and its environment. A channel may also connect the state machine (composite state) of an agent with the environment and with contained agents.

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. 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 indeterminate and non-constant time interval, the first signal instance in the queue is released and given to one of the endpoints 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.

When a signal instance is sent to an instance of the same agent instance set, interpretation of the *Output-node* either implies that the signal is put directly in the input port of the destination agent, or that the signal is sent via a channel without delay which connects the agent instance set to itself.

A remote procedure or remote variable on a channel is mentioned as outgoing from an importer and incoming to an exporter.

Model

If the <channel name> is omitted from a <channel definition> or <channel definition area>, the channel is implicitly and uniquely named.

If an agent or agent type contains explicit or implicit gates not connected by explicit channels, implicit channels without signal lists are derived. Thereafter, channels without explicit signal lists are filled in with signals, remote procedures and remote variables. The details of this are described in Annex F.

Explicitly defined <channel path>s must have non-empty signal lists after these transformations.

A channel with both endpoints being gates of one <textual typebased agent definition> represents individual channels from each of the agents in this set to all agents in the set, including the originating agent. Any resulting bidirectional channel connecting an agent in the set to the agent itself is split into two unidirectional channels.

10.2 Connection

Concrete textual grammar

```
<channel to channel connection> ::=
                        connect <external channel identifiers>
                        and <channel identifiers> <end>

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

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

No channel may be mentioned after the keyword **and** in more than one <channel to channel connection> of a given scope unit.

For any pair of <channel to channel connection>s of a given scope unit, the <external channel identifiers>s shall either mention the same set of channels, or shall have no channels in common. If two or more <channel to channel connection>s of a given scope unit have the same set of external channels, this is derived syntax for a single <channel to channel connection> having one of the <external channel identifiers> as its <external channel identifiers>, and its <channel identifiers> formed by listing all the internal <channel identifier>s.

Concrete graphical grammar

Graphically, the connect construct is represented by the graphical connection of a <channel symbol> in a <channel definition area> to an <external channel identifiers>in a <gate on diagram>.

No <channel identifier> shall be mentioned more than once in the connections of a diagram.

NOTE – Because of the **connect** construct, an (external) channel which can be anonymous in the graphical version of a specification, may need to have a name in the corresponding textual version. This is completely analogous to the case of <merge area>s in process or procedure graphs. A tool that converts the graphical version of a specification to a textual version should thus be able to generate implicit channel names.

Semantics

The <channel identifier>s in an <external channel identifiers> part of a <channel to channel connection> must denote channels connected to the enclosing agent. Each channel connected to the enclosing agent must be mentioned in the <external channel identifiers> part of at least one <channel to channel connection>.

Each channel identified by a <channel identifier> in a <channel identifiers> part of a <channel to channel connection> must be defined in the same agent in which the <channel to channel connection> is defined and it must have the boundary of that agent as one of its endpoints. Each channel defined in the surrounding agent and which has its environment as one of its endpoints, must be mentioned in the <channel identifiers> part of exactly one <channel to channel connection>.

Model

Connections are shorthand constructs and are transformed to gates.

Each different <channel to channel connection> in a given scope unit defines one implicit gate on the scope unit. All channels in the <channel to channel connection> are connected to that gate in their respective scope units. The gate constraints of the implicit gate are derived from the channels connected to the gate.

The name of the gate is a unique and unambiguous derived name. In the surrounding scope unit the <channel definition> that is identified by the <channel identifier> is extended with a <via gate> part. The <via gate> part is added to the <channel endpoint> that references the current scope unit and it mentions the implicit gate. Inside the scope unit the channels that are associated with the external channel by means of the <channel to channel connection> are modified, by extending the <channel endpoint> that mentions **env** with a <via gate> part for the implicit gate.

10.3 Signal

Abstract grammar

Signal-definition	::	*Signal-name*
		*Sort-reference-identifier**
Signal-identifier	=	*Identifier*
Signal-name	=	*Name*

Concrete textual grammar

```
<signal definition>::=
                    <type preamble>
                    signal <signal definition item> { , <signal definition item> }* <end>

<signal definition item> ::=
                    <signal name>
                    [<formal context parameters>]
                    [<virtuality constraint>]
                    [<specialization>]
                    [<sort list>]
```

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

<signal reference> ::=
    <type preamble>
    signal <signal identifier> <type reference properties>

<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>.

An abstract signal can only be used in specialization and signal constraints.

Concrete graphical grammar

<signal reference area> ::=
    <type reference area>

The <type reference area> that is part of a <signal reference area> must have a <graphical type reference heading> that contains a <signal name>.

Semantics

A signal instance is a flow of information between agents, and is an instantiation of a signal type defined by a signal definition. A signal instance can be sent by either the environment or an agent and is always directed to either an agent or the environment. A signal instance is created when an *Output-node* is interpreted and ceases to exist when an *Input-node* is interpreted.

The semantics of <virtuality> is defined in 8.3.2.

10.4 Signal list definition

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

Concrete textual grammar

<signal list definition> ::=
    signallist <signal list name> <equals sign> <signal list> <end>

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

<signal list item> ::=
    <signal identifier>
  | **(** <signal list identifier> **)**
  | <timer identifier>
  | [**procedure**] <remote procedure identifier>
  | [**interface**] <interface identifier>
  | [**remote**] <remote variable 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 and by replacing all the <remote procedure identifier>s and all the <remote variable identifier>s by one of the implicit signals each of them denotes (see 10.5 and 10.6), corresponds to a *Signal-identifier*-**set** in the *Abstract grammar*.

A <signal list item> which is an <identifier> denotes a <signal identifier> or <timer identifier> if this is possible according to the visibility rules or else a <remote procedure identifier> if this is possible according to the visibility rules, or else a <remote variable identifier>. To force a <signal list item> to denote a <remote procedure identifier>, <interface identifier> or <remote variable identifier> the keyword **procedure**, **interface** or **remote** respectively can be used.

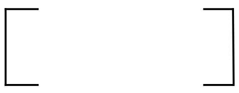
The <signal list> must not contain the <signal list identifier> defined by the <signal list definition> either directly or indirectly (via another <signal list identifier>).

It is only allowed to use **this** in <signal list>s that are part of <gate constraint>s (see 8.1.6).

Concrete graphical grammar

<signal list area> ::=
    <signal list symbol> ***contains*** <signal list>

<signal list symbol> ::=



10.5 Remote procedures

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

Concrete textual grammar

<remote procedure definition> ::=
    remote procedure <remote procedure name> [**nodelay**]
    <procedure signature> <end>

<remote procedure call> ::=
    call <remote procedure call body>

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

<communication constraints> ::=
    {**to** <destination> | **timer** <timer identifier> | <via path>}*

The use of **nodelay** is described in 10.1.

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.

A remote procedure mentioned in a <remote procedure call> must be in the complete output set (see Annex F) of an enclosing agent type or agent set.

If <destination> in a <remote procedure call body> is a <pid expression> with a sort other than Pid (see 12.1.6), then the <remote procedure identifier> must represent a remote procedure contained in the interface that defined the pid sort.

When the <destination> and the <via path> are omitted, there is a syntactic ambiguity between <remote procedure call body> and <procedure call>. In this case, the contained <identifier> denotes a <procedure identifier> if this is possible according to the visibility rules and otherwise a <remote procedure identifier>.

The <timer identifier> of <communication constraints> must not have the same <identifier> as an <exception identifier>.

In a <remote procedure call body> a <communication constraints> list is associated to the last <remote procedure identifier>. For example, in

    call p **to call** q **timer** t **via** g

the **timer** t as well as **gate** g would apply to the **call** of q.

A <communication constraints> shall contain no more than one <destination> and no more than one <timer identifier>.

Concrete graphical grammar

```
<remote procedure call area> ::=
                    <procedure call symbol> contains <remote procedure call body>
                    [ is connected to <on exception association area> ]
```

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 and an exported procedure is established by both referring to the same <remote procedure definition>.

A remote procedure call by a requesting agent causes the requesting agent to wait until the server agent has interpreted the procedure. Signals sent to the requesting agent while it is waiting are saved. The server agent will interpret 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 <save part> nor <input part> is specified for a state, an implicit transition consisting of the procedure call only and leading back to the same state is added. If an <input part> is specified for a state, an implicit transition consisting of the procedure call followed by <transition> is added. If a <save part> is specified for a state, an implicit save of the signal for the requested procedure is added.

Model

A remote procedure call

```
        call Proc(apar) to destination timer timerlist via viapath
```

is modelled by an exchange of implicitly defined signals. If the **to** or **via** clauses are omitted from the remote procedure call, they are also omitted in the following transformations. The channels are explicit if the remote procedure has been mentioned in the <signal list> (the outgoing for the importer and the incoming for the exporter) of at least one gate or channel connected to the importer or exporter. When a remote procedure is conveyed on explicit channels, the **nodelay** keyword from the <remote procedure definition> is ignored. The requesting agent sends a signal containing the actual parameters of the procedure call, except actual parameters corresponding to **out**-parameters, to the server agent and waits for the reply. In response to this signal, the server agent interprets the corresponding remote procedure, sends a signal back to the requesting agent with the results of all **in**/**out**-parameters and **out**-parameters, and then interprets 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>. Both pCALL and pREPLY have a first parameter of the predefined Integer sort.

On each channel mentioning the remote procedure, the remote procedure is replaced by *pCALL*. For each such channel, a new channel is added in the opposite direction; this channel carries the signal *pREPLY*. The new channel has the same delaying property as the original one.

a) For each imported procedure, two implicit Integer variables n and newn are defined, and n is initialized to 0.

NOTE 1 – The parameter n is introduced to recognize and discard reply signals of remote procedure calls which were left through associated timer expiry.

The <remote procedure call> is transformed as below:

```
task n:= n + 1;
output pCALL(apar,n) to destination via viapath;
wait in state pWAIT, saving all other signals;
input pREPLY(aINOUTpar,newn);
```

```
decision newn = n;
        (true):
        (false): nextstate pWAIT;
        enddecision;
        return;
```

where apar is the list of actual parameters except actual parameters corresponding to out parameters, and aINOUTpar is the modified list of actual in/out-parameters and out-parameters, including an additional parameter if a value returning remote procedure call is transformed.

NOTE 2 – The return statement terminates the implicit procedure introduced according to 11.12.1.

For every exception contained in the <raises> of a remote procedure p and all predefined exceptions e a signal eRAISE is defined which can transport all exception parameters of e. The following will be inserted into the state pWAIT:

```
state pWAIT;
input eRAISE(params,newn);
decision newn = n;
(true): raise e(params);
(false): nextstate pWAIT;
enddecision;
```

For a timer t included in <communication constraints> an additional exception with the same name and the same parameters is implicitly inserted in the same scope as the timer definition, and there must not be an explicitly defined exception with the same name as the timer in the same scope unit where the timer is defined.

Additionally, the following will be inserted for a timer t that is included in <communication constraints>

```
state pWait;
input t(aParams);
raise t(aParams);
```

where aParams stands for implicitly defined variables with the sort of the parameters contained in the timer definition.

In all states of the agent except pWAIT

```
input pReply, eRAISE;
nextstate actual state;
```

is inserted.

b) In the server agent, an implicit exception handler pEXC and an implicit Integer variable n is defined for each explicit or implicit <input part> being a remote-procedure input. Furthermore, there is one ivar variable for each such <input part> defined in the scope where the explicit or implicit remote procedure input occurs. If a value returning remote procedure call is transformed, an implicit variable res with the same sort as <sort> in <procedure result> is defined.

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

```
input pCALL(fpar,n);
task ivar:= sender;
call Proc(fpar); onexception pEXC;
output pREPLY(INOUTpar,n) to ivar;
transition;
```

or,

```
input pCALL(fpar,n);
task ivar:= sender;
task res := call Proc(fpar); onexception pEXC;
output pREPLY(INOUTpar,res,n) to ivar;
transition;
```

if a value returning remote procedure call was transformed.

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

```
save pCall;
```

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

```
input pCALL;
output eRAISE(params,n) to sender;
transition;
```

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

```
input pCALL(fpar,n);
task ivar:= sender;
call Proc(fpar); onexception pEXC;
output pREPLY(INOUTpar,n) to ivar;
/* next state the same */
```

For every exception e contained in the <raises> of the remote procedure, and for every predefined exception the following is inserted:

```
exceptionhandler pEXC;
handle e(params);
output eRAISE(params,n) to ivar;
raise e(params);
```

If an exception handler is associated to a remote-procedure input, the exception handler becomes associated to the resulting signal input (not shown in the model above).

NOTE 3 – 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 an agent which is guaranteed by the specification to exist at the time of receiving the pCALL signal. Associated timers allow the deadlock to be avoided.

10.6 Remote variables

In SDL, a variable is always owned by, and local to, an agent instance. Normally the variable is visible only to the agent instance that owns it and to the contained agents. If an agent instance in another agent needs to access the data items associated with a variable, a signal interchange with the agent instance owning the variable is needed.

This can be achieved by the following shorthand notation, called imported and exported variables. The shorthand notation may also be used to export data items to other agent instances within the same agent.

Concrete textual grammar

```
<remote variable definition> ::=
        remote <remote variable name> {,<remote variable name>}* <sort> [ nodelay ]
                {, <remote variable name> {, <remote variable name>}* <sort> [ nodelay ]}*
                <end>

<import expression> ::=
        import ( <remote variable identifier> <communication constraints> )

<export> ::=
        export ( <variable identifier> { , <variable identifier> }* )
```

The use of **nodelay** is described in Annex F.

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.

A remote variable mentioned in an <import expression> must be in the complete output set (see Annex F) of an enclosing agent type or agent set.

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

If <destination> in an <import expression> is a <pid expression> with a sort other than Pid (see 12.1.6), then the <remote variable identifier> must represent a remote variable contained in the interface that defined the pid sort.

There is no corresponding graphical syntax for <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 and an exported variable is established by both referring to the same <remote variable definition>.

Imported variables are specified as part of the output set of the enclosing active entity. Exported variables are specified as part of the complete input set of the enclosing active entity.

The agent instance that owns a variable whose data items are exported to other agent instances is called the exporter of the variable. Other agent instances that use these data items are known as importers of the variable. The variable is called exported variable.

An agent instance may be both importer and exporter of the same remote variable.

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 interpretation of an <export> by which an exporter discloses the current result of an exported variable. An export operation causes the storing of the current result of the exported variable into its implicit copy.

b) *Import operation*

An import operation is the interpretation of an <import expression> by which an importer accesses the result of an exported variable. The result is stored in an implicit variable denoted by the <remote variable 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 agent 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 referring to the same remote variable in the export variable definition and in the <import expression>.

Model

An import operation is modelled by exchange of implicitly defined signals. When a remote variable is conveyed on explicit channels, the **nodelay** keyword from the <remote variable definition> is ignored. 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 result 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 result 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 signal xQUERY has an argument of the predefined sort Integer and xREPLY has arguments of the sort of the variable and Integer. The implicit copy of the exported variable is denoted by imcx.

On each channel mentioning the remote variable, the remote variable is replaced by xQUERY. For each such channel, a new channel is added in the opposite direction; this channel carries the signal xREPLY. In case of a channel, the new channel has the same delaying property as the original one.

For each predefined exception (denoted as predefExc), an additional anonymous signal (denoted as predefExcRAISE) is defined.

a) *Importer*

For each imported variable, two implicit Integer variables n and newn are defined, and n is initialized to 0. In addition, an implicit variable x of the sort of the remote variable is defined.

The <import expression>

```
import (x to destination via via-path)
```

is transformed to the following, where the **to** clause is omitted if the destination is not present, and the **via** clause is omitted if it is not present in the original expression:

```
task n:= n + 1;
output xQUERY(n) to destination via via-path;
wait in state xWAIT, saving all other signals;
input xREPLY(x,newn);
decision newn = n;
(true):
(false): nextstate xWAIT;
enddecision;
return;
state xWAIT
input predefExcRAISE;
raise predefExc;
```

In all other states, xREPLY is saved.

NOTE 1 – The return statement terminates the implicit procedure introduced according to 11.12.1.

For every timer t included in <communication constraints>, an additional exception with the same name and the same parameters is implicitly inserted in the same scope as the timer definition. In that case there must not be an exception with the same name in the scope unit of the timer definition.

Additionally, the following will be inserted for every timer t that is included in <communication constraints>:

```
state xWait;
input t(aParams);
raise t(aParams);
```

where aParams stand for implicitly defined variables with the sort of the parameters contained in the timer definition.

The result of the transformation is encapsulated in an implicit procedure, as described in 11.12.1. Every <on exception> attached to the import action shall be attached to a call of the implicit procedure.

b) *Exporter*

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

```
input xQUERY(n);
task ivar := sender;
output xREPLY(imcx,n) to ivar; onexception xEXC;
nextstate the state containing this input;
exceptionhandler xEXC;
handle predefExc;
output predefExcRAISE to ivar;
raise predefExc;
```

For each such state, ivar will be defined as variable of sort Pid, and n as a variable of type Integer.

The <export>

```
    export x
```

is transformed to the following:

```
    task imcx := x;
```

NOTE 2 – 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 an agent which is guaranteed by the specification to exist at the time of receiving the xQUERY signal. Specifying a set timer in the <import expression> avoids such a deadlock.

11 Behaviour

11.1 Start

Abstract grammar

State-start-node	::	[*On-exception*]
		[*State-entry-point-name*]
		Transition

Concrete textual grammar

<start> ::=

start [<virtuality>] [<state entry point name>] <end> [<on exception>] <transition>

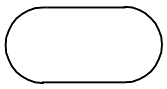
If <state entry point name> is given in a <start>, the <start> must be the <start> of a <composite state>.

Concrete graphical grammar

<start area> ::=

<start symbol> ***contains*** { [<virtuality>] [<state entry point name>]}
[***is connected to*** <on exception association area>]
is followed by <transition area>

<start symbol> ::=



If <state entry point name> is given in a <start>, the <start area> must be the <start area> of a <composite state>.

Semantics

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

Model

A <start> or <start area> which contains <virtuality> is called a virtual start. Virtual start is further described in 8.3.3.

A <start> or <start area> contained in a <composite state> or <composite state area> is defined in 11.11.

11.2 State

Abstract grammar

State-node	::	*State-name*
		[*On-exception*]
		Save-signalset
		*Input-node-**set***
		*Spontaneous-transition-**set***
		*Continuous-signal-**set***
		[*Composite-state-type-identifier*]
State-name	=	*Name*

*State-node*s within a *State-transition-graph* or *Procedure-graph* must have different *State-name*s.

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

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

A *State-node* with *Composite-state-type-identifier* represents a <composite state application>.

Concrete textual grammar

<state> ::=
        <basic state> | <composite state application>

A <composite state application> represents a state with a *Composite-state-type-identifier.*

Concrete graphical grammar

<state area> ::=
        <basic state area> | <composite state application area>

A <composite state area> represents a state with a *Composite-state-type-identifier.*

11.2.1 Basic State

Concrete textual grammar

<basic state> ::=
        state <state list> <end> [<on exception>]
            {   <input part>
            | <priority input>
            | <save part>
            | <spontaneous transition>
            | <continuous signal> }*
        [**endstate** [<state name>] <end>]

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

<asterisk state list> ::=
        <asterisk> [**(** <state name> { **,** <state name>}* **)**]

A given state may have at most one exception handler associated.

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>.

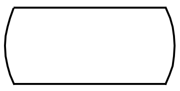
A <state name> may appear in more than one <state> of a body.

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.

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>.

Concrete graphical grammar

```
<basic state area> ::=
                    <state symbol> contains <state list>
                    [ is connected to <on exception association area> ]
                    is associated with
                         {     <input association area>
                         |     <priority input association area>
                         |     <continuous signal association area>
                         |     <spontaneous transition association area>
                         |     <save association area> }*

<state symbol> ::=
```



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

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

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

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

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

Semantics

A state represents a particular condition in which the state machine of an agent may consume a signal instance. If a signal instance is consumed, the associated transition is interpreted. A transition may also be interpreted as the result of a continuous signal or a spontaneous transition.

For each state, the *Save-signal*s, *Input-node*s, *Spontaneous-signal*s, and *Continuous-signal*s are interpreted in the following order:

a) if the input port contains a signal matching a priority input of the current state, the first such signal is consumed (see 11.4); otherwise

b) in the order of the signals on the input port:
   1) the *Provided-expression*s of the *Input-node* corresponding to the current signal are interpreted in arbitrary order, if any;
   2) if the current signal is enabled, this signal is consumed (see 11.6); otherwise
   3) the next signal on the input port is selected.

c) if no enabled signal was found, in priority order of the *Continuous-signal*s, if any, with *Continuous-signal*s of equal priority being considered in an arbitrary order and no priority being treated as the lowest priority:
   1) the *Continuous-expression* contained in the current *Continuous-signal* is interpreted;
   2) if the current continuous signal is enabled, this signal is consumed (see 11.5); otherwise
   3) the next continuous signal is selected.

d) if no enabled signal was found, the state machine waits in the state until another signal instance is received. If the state has enabling conditions or continuous signals, these steps are repeated even if no signal is received.

At any time in a state which contains *Spontaneous-transition*s, the state machine may interpret the *Provided-expression* of a *Spontaneous-transition* and subsequently, if the *Spontaneous-transition* was enabled, the *Transition* of one of the *Spontaneous-transition*s (see 11.9), or the *Transition* of one of the *Spontaneous-transition*s, if there was no *Provided-expression*.

Model

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

When several <state>s contain the same <state name>, these <state>s are concatenated into one <state> having that <state name>.

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>.

11.2.2 Composite state application

A <composite state application> describes that the state machine has a composite state. The properties of the composite state are defined either as part of the <composite state application>, by a referenced composite state, or by a composite state type.

Concrete textual grammar

```
<composite state application> ::=
                        state <composite state list> <end> [<on exception>]
                            {     <input part>
                            |     <priority input>
                            |     <save part>
                            |     <spontaneous transition>
                            |     <continuous signal>
                            |     <connect part> }*
                        [ endstate [ <state name> ] <end> ]
```

A <composite state reference> to the same composite state must only occur in one of the <composite state application>s in the surrounding state machine.

A *State-node* with *Composite-state-type-identifier* represents a <composite state application>.

Concrete graphical grammar

```
<composite state application area> ::=
                        <state symbol> contains <composite state list>
                        [ is connected to <on exception association area> ]
                        is associated with
                            {     <input association area>
                            |     <priority input association area>
                            |     <continuous signal association area>
                            |     <spontaneous transition association area>
                            |     <save association area>
                            |     <connect association area> }*

<connect association area> ::=
                        <solid association symbol> is connected to <connect area>

<composite state list> ::=
                        {<state list token> { , <state list token> }* }
                  |     {<asterisk state list>}

<state list token> ::=
                        <state name> [<actual parameters>]
                  |     <typebased composite state>
```

A <state list token> must only contain <actual parameters> if <state area> coincides with a <nextstate area>. In this case the <state area> must only contain one <state name> and, optionally, <actual parameters>.

Semantics

The semantics for <composite state application> is defined in 11.11.

11.3 Input

Abstract grammar

```
Input-node                    ::   [PRIORITY]
                                   Signal-identifier
                                   [Variable-identifier]*
                                   [Provided-expression]
                                   [On-exception]
                                   Transition
Variable-identifier           =    Identifier
```

The length of the list of optional *Variable-identifier*s 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 data items that can be carried by the signal.

Concrete textual grammar

```
<input part> ::=
                    input [<virtuality>] <input list> <end>
                    [<on exception>] [<enabling condition>] <transition>

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

<stimulus> ::=
                    <signal list item>
                    [ ( [ <variable> ] { , [ <variable> ] }* ) | <remote procedure reject> ]

<remote procedure reject> ::=
                    raise <exception raise>

<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>.

A <remote procedure reject> may be specified only if the <signal list item> denotes a <remote procedure identifier>. The <exception identifier> in the <remote procedure reject> must be mentioned in the <remote procedure definition>.

A <signal list item> must not denote a <remote variable identifier> and if it denotes a <remote procedure identifier> or a <signal list identifier>, the <stimulus> parameters (including the parenthesis) must be omitted.

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 11.15.

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

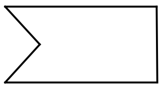
In the *Abstract Grammar*, the <remote procedure identifier>s are also represented as *Signal-identifier*s.

Concrete graphical grammar

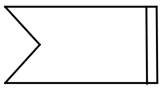
<input area> ::=
  <input symbol> ***contains*** { [<virtuality>] <input list> }
  [***is connected to*** <on exception association area>]
  [***is associated with*** <solid association symbol> ***is connected to*** <enabling condition area>]
  is followed by <transition area>

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

<plain input symbol> ::=



<internal input symbol> ::=



An <input area> whose <input list> contains one <stimulus> corresponds to one *Input-node*. Each of the <<u>signal</u> identifier>s or <<u>timer</u> identifier>s contained in an <input symbol> gives the name of one of the *Input-node*s which this <input symbol> represents.

NOTE – There is no difference in meaning between a <plain input symbol> and an <internal input symbol>.

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 agent. The variables associated with the input are assigned the data items conveyed by the consumed signal.

The data items are 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 corresponding data item is discarded. If there is no data item associated with a sort specified in the signal, the corresponding variable becomes "undefined".

The sender of the consuming agent (see clause 9, *Model*) is given the pid of the originating agent, as carried by the signal instance.

Signal instances flowing from the environment to an agent instance within the system will always carry a pid different from any in the system.

Model

A <stimulus> whose <signal list item> is a <<u>signal list</u> identifier> is derived syntax for a list of <stimulus>s without parameters and is inserted in the enclosing <input list> or <priority input list>. In this list, there is a one to one correspondence between the <stimulus>s and the members of the signal list.

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

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 <agent definition>, except for <<u>signal</u> identifier>s of implicit input signals introduced by the concepts in 10.5, 10.6, 11.4, 11.5 and 11.6 and for <signal identifier>s contained in the other <input list>s and <save list>s of the <state>.

An <input part> or <input area> that contains <virtuality> is called a virtual input transition. Virtual input transition is further described in 8.3.3.

11.4 Priority Input

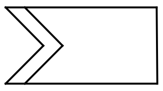
In some cases, it is convenient to express that reception of a signal takes priority over reception of other signals. This can be expressed by means of priority input.

Concrete textual grammar

```
<priority input> ::=
                    priority input [<virtuality>]
                    <priority input list> <end> [<on exception>]<transition>

<priority input list> ::=
                    <stimulus> {, <stimulus>}*
```

Concrete graphical 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 connected to <on exception association area> ]
                    is followed by <transition area>

<priority input symbol> ::=
```



A <priority input> or <priority input association area> represents an *Input-node* with **PRIORITY**.

Semantics

If an *Input-node* of a state has **PRIORITY**, the signal is a priority signal and will be consumed before any other signals are consumed, provided it has an enabled transition.

Model

A <priority input> or <priority input area> which contains <virtuality> is called a virtual priority input. Virtual priority input is further described in 8.3.3.

11.5 Continuous signal

In describing systems, the situation may arise where a transition should be interpreted when a certain condition is fulfilled. A continuous signal interprets a Boolean expression and the associated transition is interpreted when the expression returns the predefined Boolean value true.

Abstract grammar

Continuous-signal	::	*Continuous-expression* [*Priority-name*] *Transition*
Continuous-expression	=	*Boolean-expression*
Priority-name	=	*Nat*

Concrete textual grammar

```
<continuous signal> ::=
                    provided [<virtuality>]
                          <continuous expression> <end>
                    [ priority <priority name> <end> ] [<on exception>] <transition>
```

<continuous expression> ::=
    <Boolean expression>

<priority name> ::=
    <Natural literal name>

Concrete graphical grammar

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

<continuous signal area> ::=
    <enabling condition symbol>
    contains {
        [<virtuality>] <continuous expression>
        [[<end>] **priority** <priority name>] }
    [***is connected to*** <on exception association area>]
    is followed by <transition area>

Semantics

The *Continuous-expression* is interpreted upon entering the state to which its *Continuous-signal* is associated, and while waiting in the state, whenever no <stimulus> of an attached <input list> is found in the input port. If the *Continuous-expression* returns the predefined Boolean value true, the continuous signal is enabled.

The continuous signal having the lowest value for *Priority-name* has the highest priority.

Model

A <continuous signal> or <continuous signal area> that contains <virtuality> is called a virtual continuous signal. Virtual continuous transition is further described in 8.3.3.

11.6 Enabling condition

An enabling condition makes it possible to impose an additional condition on the consumption of a signal, beyond its reception as well as on a spontaneous transition.

Abstract grammar

Provided-expression = *Boolean-expression*

Concrete textual grammar

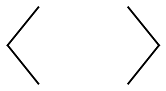
<enabling condition> ::=
    provided <provided expression> <end>

<provided expression> ::=
    <Boolean expression>

Concrete graphical grammar

<enabling condition area> ::=
    <enabling condition symbol> ***contains*** <provided expression>

<enabling condition symbol> ::=



When the <provided expression> contains an <imperative expression>, then *Provided-expression* is a *Value-returning-call-node* containing only the *Procedure-identifier* of the procedure implicitly defined by the *Model* below. Otherwise, *Provided-expression* is represented by <provided expression>.

Semantics

The *Provided-expression* of an *Input-node* is interpreted when entering the state this *Input-node* is attached to, and any time the state is re-entered through the arrival of a <stimulus>. In the case of multiple enabling conditions, these are interpreted sequentially in an arbitrary order when entering the state.

A signal in the input port is enabled, if all the *Provided-expression*s of an *Input-node* return the predefined Boolean value true, or if the *Input-node* does not have a *Provided-expression*. The *Provided-expression* of a *Spontaneous-transition* can be interpreted at any time while the agent is in the state.

Model

When the <provided expression> contains an <imperative expression>, the following procedure with an anonymous name referred to as *isEnabled* is implicitly defined.

```
procedure isEnabled -> Boolean;
    start ;
    return <provided expression>;
endprocedure ;
```

NOTE – The <Boolean expression> may be further transformed according to the model of <import expression>.

11.7 Save

A save specifies a set of signal identifiers and remote procedure identifiers whose instances are not relevant to the agent 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

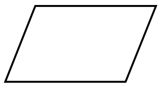
```
<save part> ::=
                    save [<virtuality>] <save list> <end>

<save list> ::=
                    <signal list>
                |   <asterisk save list>

<asterisk save list> ::=
                    <asterisk>
```

A <save list> represents the *Signal-identifier-**set***.

A <state> may contain at most one <asterisk save list>. A <state> must not contain both <asterisk input list> and <asterisk save list>.

Concrete graphical grammar

```
<save area> ::=
                    <save symbol> contains { [<virtuality>] <save list> }

<save symbol> ::=
```



Semantics

A signal in a Save-signalset is not enabled.

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

An <asterisk save list> is transformed to a list of <stimulus>s containing the complete valid input signal set of the enclosing <agent definition>, except for <signal identifier>s of implicit input signals introduced by the concepts in 10.5, 10.6, 11.4, 11.5 and 11.6 and for <signal identifier>s contained in the other <input list>s and <save list>s of the <state>.

A <save part> or <save area> which contains <virtuality> is called a virtual save. Virtual save is further described in 8.3.3.

11.8 Implicit transition

Concrete textual grammar

A <signal identifier> contained in the complete valid input signal set of an <agent 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> that only contains a <nextstate> leading back to the same <state>.

11.9 Spontaneous transition

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

Abstract grammar

Spontaneous-transition	::	[*On-exception*]
		[*Provided-expression*]
		Transition

Concrete textual grammar

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

<spontaneous designator> ::=
    none

Concrete graphical grammar

<spontaneous transition area> ::=
    <input symbol> ***contains*** { [<virtuality>] <spontaneous designator> }
    [***is connected to*** <on exception association area>]
    [***is associated with*** <solid association symbol> ***is connected to*** <enabling condition area>]
    is followed by <transition area>

Semantics

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

After activation of a spontaneous transition, the **sender** expression of the agent returns **self**.

Model

A <spontaneous transition> or <spontaneous transition area> that contains <virtuality> is called a virtual spontaneous transition. Virtual spontaneous transition is further described in 8.3.3.

11.10 Label

Abstract grammar

Free-action	::	*Connector-name*
		Transition
Connector-name	=	*Name*

Concrete textual grammar

```
<label> ::=
                    <connector name> :

<free action> ::=
                    connection
                        <transition>
                    [ endconnection [ <connector name> ] <end> ]
```

The term "body" is used to refer to a state machine graph, possibly after transformation from a <statement list> and after transformation from a type. A body therefore refers to <agent body>, <procedure body>, <operation body>, <state machine graph area>, <procedure graph area>, <operation graph area>, or <composite state body>.

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

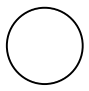
A label represents the entry point of a transfer of control from the corresponding joins with the same <connector name>s in the same body.

Transfer of control is only allowed to labels within the same body. It is permissible to have a join from the body of the specialization to a connector defined in the supertype.

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 <state machine graph area> or <type state machine graph area> or <procedure graph area>.

Semantics

A *Free-action* defines the target of a *Join-node*. In the abstract grammar, only free actions have labels; labels inside of a transition are transformed into separate free actions.

Model

If a <label> is not the first label of a <transition string>, the <transition string> is split into two parts. All <action statement>s preceding the <label> are preserved in the original transition, which is terminated with a <join> to the <label>. All action statements following <label> are copied to a new <free action>, which starts with the <label>.

11.11 State machine and Composite state

A composite state is a state that may either consist of sequentially interpreted substates (with associated transitions), or of an aggregation of substates interpreted in an interleaving mode. A substate is a state, so a substate may in turn be a composite state.

The properties of a composite state (substates, transitions, variables, and procedures) are defined by a <composite state> or a <composite state type definition>, and the specification of a <state> with <composite state name> within a state machine or a composite state. Transitions associated with a composite state apply to all substates of the composite state.

Abstract grammar

Composite-state-formal-parameter	=	*Agent-formal-parameter*
State-entry-point-definition	=	*Name*
State-exit-point-definition	=	*Name*
Entry-procedure-definition	=	*Procedure-definition*
Exit-procedure-definition	=	*Procedure-definition*
Named-start-node	::	*State-entry-point-name*
		[*On-exception*]
		Transition
State-entry-point-name	=	*Name*

Entry-procedure-definition represents a procedure with the name entry. *Exit-procedure-definition* represents a procedure with the name exit. These procedures may not have parameters, and may only contain a single transition.

Concrete textual grammar

```
<composite state> ::=
                <composite state graph> | <state aggregation>
```

Concrete graphical grammar

```
<composite state area> ::=
                <composite state graph area> | <state aggregation area>
```

Semantics

A composite state is created when the enclosing entity is created, and deleted when the enclosing entity is deleted.

Local variables will be created and deleted when the composite state is created and deleted respectively. If a <variable definition> contains a <constant expression>, the <variable definition> is assigned the result of the <constant expression> at creation time. If no <constant expression> is present, the result of the <variable definition> is undefined.

Composite-state-formal-parameters are local variables that are created when the composite state is created. A variable gets the result of the expression given by the corresponding actual parameter if present in the *Nextstate-node*. Otherwise, the result of the variable becomes undefined.

A transition emanating from a substate has higher priority than a conflicting transition emanating from any of the containing states. Conflicting transitions are transitions triggered by the same input, priority input, save or continuous signal.

Entry-procedure-definition and *Exit-procedure-definition*, if defined, are called implicitly when the state is entered and exited, respectively. It is not mandatory to define both procedures. The entry procedure is called before the start transition is invoked, or if the state is re-entered as a result of interpreting a *Nextstate-node* with **HISTORY**. The exit procedure is invoked after a *Return-node* of the *Composite-state-graph* is interpreted, or when a transition attached directly to the *State-node* is interpreted. When an exception is raised in a composite state, the exit procedure is not invoked.

11.11.1 Composite State Graph

In a composite state graph, the transitions are interpreted sequentially.

Abstract grammar

```
Composite-state-graph           ::    State-transition-graph
                                      [Entry-procedure-definition]
                                      [Exit-procedure-definition]
                                      Named-start-node-set
```

Concrete textual grammar

```
<composite state graph> ::=
                    {<package use clause>}*
                    <composite state heading> <end> substructure
                         [<valid input signal set>]
                         {<gate in definition>}*
                         <state connection points>*
                         <entity in composite state>*
                         <composite state body>
                    endsubstructure [ [<qualifier>] <composite state name> ] <end>

<composite state heading> ::=
                    state [<qualifier>] <composite state name>
                    [<agent formal parameters>]

<entity in composite state> ::=
                    <variable definition>
               |    <data definition>
               |    <select definition>
               |    <data type reference>
               |    <macro definition>
               |    <procedure definition>
               |    <exception definition>
               |    <composite state type definition>
               |    <composite state type reference>

<composite state body> ::=
                    [<on exception>] <start>* { <state> | <exception handler> | <free action> }*

<composite state reference> ::=
                    state substructure <composite state name> referenced <end>
```

Composite-state-graph represents <composite state body>.

If a <composite state body> contains at least one <start> but no <state>, the <composite state body> shall be interpreted as an encapsulated part of a transition.

Exactly one of the <start>s shall be unlabelled. Each additional labelled entry and exit point must be defined by a corresponding <state connection points>.

If a <composite state body> contains at least one <state> different from asterisk state, a <start> must be present.

<variable definition> in a <composite state> cannot contain **exported** <variable name>s, if the <composite state> is enclosed by a <procedure definition>.

Concrete graphical grammar

```
<composite state graph area> ::=
                    <frame symbol> contains {
                        <composite state heading>
                        {     <composite state text area>*
                              <type in composite state area>*
                              <composite state body area> } set }
                    is associated with {<graphical state connection point>* } set
                    is connected to { {<gate on diagram>}* }set
                    [ is associated with <package use area> ]

<composite state text area> ::=
                    <text symbol> contains
                    {     <valid input signal set>
                    |     <variable definition>
                    |     <data definition>
                    |     <data type reference>
                    |     <procedure definition>
                    |     <procedure reference>
                    |     <select definition>
                    |     <macro definition>}*

<type in composite state area> ::=
                    <procedure area>
                |   <data type reference area>
                |   <composite state type diagram>
                |   <composite state type reference area>

<composite state body area> ::=
                    [<on exception association area>]
                    <start area>* { <state area> | <exception handler area> | <in connector area> }*
```

There shall be at most one <valid input signal set> in the <composite state text area>s of a <composite state graph area>.

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

Semantics

The unlabelled *State-start-node* of the *Composite-state-graph* is interpreted as the default entry point of the composite state. It is interpreted when the *Nextstate-node* has no *State-entry-point*. *Named-start-nodes* are interpreted as additional entry points of the composite state. The *State-entry-point* of a *Nextstate-node* defines which named start transition is interpreted.

An *Action-return-node* in a composite state (with no <state exit point name>) is interpreted as the default exit point of the composite state. Interpretation of an *Action-return-node* triggers the *Connect-node* without a *Name* in the enclosing entity. Additional *Named-return-nodes*, that is, <return> with <state exit point name>, shall be interpreted as additional exit points of the composite state. Interpretation of a *Named-return-node* will trigger an exit transition in the enclosing entity contained in a *Connect-node* with the same *Name*.

The nodes of the state graph are interpreted in the same manner as the equivalent nodes of an agent or procedure graph. That is, the state graph has the same complete valid input signal set as the enclosing agent, and the same input port as the instance of the enclosing agent.

It is possible to specify a <composite state> that only consists of transitions received in asterisk state, without <start> and without any substates. Such a <composite state> is interpreted as a basic state containing encapsulated transitions. Encapsulated transitions may either be terminated by <dash nextstate> or by <return>.

Model

If the <composite state> consists of no <state>s with <state name>s but only a <state> with <asterisk>, transform the asterisk state into a <state> with an anonymous <state name> and a <start> leading to this <state>.

11.11.2 State aggregation

A state aggregation is a partitioning of a composite state. It consists of multiple composite states, which have an interpretation that is interleaved at the transition level. At any given time each partition of a state aggregation is in one of the states of that partition, or (for one of the partitions only) in a transition, or has completed and is waiting for other partitions to complete. Each transition runs to completion. State aggregations can be used to partition the graph of a state.

Abstract grammar

State-aggregation-node	::	*State-partition-**set***
		[*Entry-procedure-definition*]
		[*Exit-procedure-definition*]
State-partition	::	*Name*
		Composite-state-type-identifier
		*Connection-definition-**set***
Connection-definition	::	*Entry-connection-definition* | *Exit-connection-definition*
Entry-connection-definition	::	*Outer-entry-point Inner-entry-point*
Outer-entry-point	::	*State-entry-point-name* | **DEFAULT**
Inner-entry-point	::	*State-entry-point-name* | **DEFAULT**
Exit-connection-definition	::	*Outer-exit-point Inner-exit-point*
Outer-exit-point	::	*State-exit-point-name* | **DEFAULT**
Inner-exit-point	::	*State-exit-point-name* | **DEFAULT**

The *State-entry-point-name* in the *Outer-entry-point* must denote a *State-entry-point-definition* of the *Composite-state-type-definition* where the *State-aggregation-node* occurs. The *State-entry-point-name* of the *Inner-entry-point* must denote a *State-entry-point-definition* of the composite state in the *State-partition*. Likewise, the *State-exit-point*s must denote exit points in the inner and outer composite state, respectively. **DEFAULT** indicates the unlabelled entry and exit points.

All entry and exit points of both the container state and the state partitions must appear in exactly one *Connection-definition*.

The input signal sets of the *State-partition-**set*** within a composite state must be disjoint. The input signal set of a *State-partition* is defined as the union of all signals appearing in an *Input-node* or the *Save-signalset* inside the composite state type, including nested states, and procedures mentioned in *Call-node*s.

Concrete textual grammar

```
<state aggregation> ::=
                    {<package use clause>}*
                    <state aggregation heading> <end> substructure
                         <state connection points>*
                         <entity in composite state>*
                         <state aggregation body>
                    endsubstructure [ [<qualifier>] <composite state name> ] <end>

<state aggregation heading> ::=
                    state aggregation [<qualifier>] <composite state name>
                    [<agent formal parameters>]

<state aggregation body> ::=
               {    <state partitioning>
               |    <state partition connection> }+
```

<state partitioning> ::=
        <textual typebased state partition definition>
    | <composite state reference>
    | <composite state>

<state partition connection> ::=
        connect <outer entry point> **and** <inner entry point> <end>

<outer entry point> ::=
        <state partition identifier> **via** <point>

<inner entry point> ::=
        <state partition identifier> **via** <point>

<point> ::=
        <state entry point> | <state exit point> | **default**

Concrete graphical grammar

<state aggregation area> ::=
        <frame symbol> ***contains*** {
            <state aggregation heading>
            { <composite state text area>*
                <type in composite state area>*
                <state aggregation body area> } ***set*** }
        is associated with {<graphical state connection point>* } ***set***
        is connected to { {<gate on diagram>}* }***set***
        [***is associated with*** <package use area>]

<state aggregation body area> ::=
        { { <state partition area> | <connection definition area>}+ }***set***

<state partition area> ::=
        <state partition reference area>
    | <composite state area>
    | <graphical typebased state partition definition>
    | <inherited state partition definition>

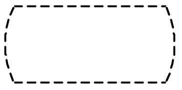
<state partition reference area> ::=
        <state symbol> ***contains*** <state name>

<graphical typebased state partition definition> ::=
        <state symbol> ***contains*** <typebased state partition heading>

<typebased state partition heading> ::=
        <state name> <colon> <composite state type expression>

<inherited state partition definition> ::=
        <dashed state symbol> ***contains*** <state partition identifier>

<dashed state symbol> ::=



<connection definition area> ::=
        <solid association symbol> ***is connected to*** <graphical point>

<graphical point> ::=
        <graphical state connection point>
    | { <state entry points> | <state exit points> } ***is associated with*** <state partition area>
    | <state connection point symbol> ***is connected to*** <frame symbol>

The same state exit point of a state partition must not be connected to more than one state exit point of the enclosing state.

One state entry point of the enclosing state must not be connected to two different state entry points of the same state partition.

Semantics

If a *Composite-state-type-definition* contains a *State aggregation-node*, the composite states of each *State-partition* are interpreted in an interleaving manner at the transition level. Each transition runs to completion before another transition is interpreted. The creation of a composite state with state partition implies the creation of the contained *State-partition-**set*** and their connections. If the *Composite-state-type-definition* of a *State-partition* has *Composite-state-formal-parameter*s, these formal parameters are *undefined* when the state is entered.

The unlabelled *State-start-nodes* of the partitions are interpreted in any order as the default entry point of the composite state. They are interpreted when the *Nextstate-node* has no *State-entry-point*. *Named-start-nodes* are interpreted as additional entry points of the composite state. If the composite state is entered through the *Outer-entry-point* of *Entry-connection-definition*s, the start transition of the partition with the corresponding *Inner-entry-point* is interpreted. The state partitions are entered in an undetermined order, after the entry procedure of the state aggregation is completed.

When each and every partition has interpreted (in any order) an *Action-Return-node* or *Named-return-node*, the partitions exit the composite state. The *Exit-connection-definitions* associate the exit points from the partitions with the exit points of the composite state. If different partitions exit the composite state through different exit points, the exit point of the composite state is chosen in a non deterministic way. The exit procedure of the state aggregation is interpreted after all state partitions have been completed. Signals in the input set of a partition that completed its return node are saved until all other partitions have been completed.

The nodes of the state partition graphs are interpreted in the same manner as the equivalent nodes of an agent, or procedure graph, with the only difference that they have disjoint input signal sets. The state partitions share the same input port as the enclosing agent.

An input transition associated with a composite state application containing a *State-aggregation-node* applies to all states of all state partitions, and it implies a default termination of all these. If such a transition terminates with a *Nextstate-node* with **HISTORY**, all partitions re-enter into their respective substates.

Model

If an entry point of the state aggregation is not connected to any entry point of a state partition, an implicit connection to the unlabelled entry is added. Likewise, if an exit point of a partition is not connected to any exit point of the state aggregation, a connection to the unlabelled exit is added.

If there are signals in the complete valid input set of an agent which are not consumed by any state partition of a certain composite state, an additional implicit state partition is added to that composite state. This implicit partition has only an unlabelled start transition and a single state containing all implicit transitions (including those for exported procedures and exported variables). When one of the other partition exits, an implicit signal is sent to the agent, which is consumed by the implicit partition. After the implicit partition has consumed all the implicit signals, it exits through a *State-return-node*.

11.11.3 State connection point

State connection points are defined in composite states, both directly specified composite states and state types, and represent connection points for entry and exit of a composite state.

Concrete textual grammar

```
<state connection points> ::=
                        { in <state entry points> | out <state exit points> } <end>

<state entry points> ::=
                        <state entry point>
                  |     ( <state entry point> { , <state entry point> }* )
```

<state exit points> ::=
<state exit point>
| **(** <state exit point> { **,** <state exit point> }* **)**

<state entry point> ::=
<state entry point name>

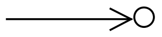
<state exit point> ::=
<state exit point name>

Concrete graphical grammar

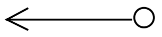
<graphical state connection point> ::=
<state connection point symbol>
is associated with { <state entry points> | <state exit points> }
is connected to <frame symbol>

<state connection point symbol> ::=
<state connection point symbol 1> | <state connection point symbol 2>

<state connection point symbol 1> ::=



<state connection point symbol 2> ::=



For <state connection point symbol 1>, the <graphical state connection point> must contain <state entry points>; otherwise the <graphical state connection point> must contain <state exit points>.

In <state connection point symbol 1> and <state connection point symbol 2>, the centre of the circle must be placed on the edge of the <frame symbol> to which it is connected.

Semantics

A <state entry point> defines an entry point on a <composite state>. A <state exit point> defines an exit point on a <composite state>.

Each <composite state> has implicitly defined two anonymous <state connection points>. These are the default entry and exit point that correspond to an unlabelled <start> and <return> respectively.

A <composite state> may only refer to its own <state entry point>s in labelled <start>s, and own <state exit point>s in labelled <return>s.

11.11.4 Connect

Abstract grammar

Connect-node	::	[*State-exit-point-name*]
		[*On-exception*]
		Transition
State-exit-point-name	=	*Name*

Concrete textual grammar

<connect part> ::=
connect [<virtuality>] [<connect list>] <end>
[<on exception>] <exit transition>

<connect list> ::=
<state exit point name> { **,** <state exit point name>}*
| <asterisk connect list>

<exit transition> ::=
<transition>

<asterisk connect list> ::=
<asterisk>

A <connect part> with at most one <state exit point> represents a *Connect-node*. If no <connect list> is given, the *State-exit-point-name* is omitted.

The <connect list> must only refer to visible <state exit point>s.

Concrete graphical grammar

```
<connect area> ::=
                    [<virtuality>] [<connect list>]
                    [ is connected to <on exception association area> ]
                    is followed by <exit transition area>

<exit transition area> ::=
                    <transition area>
```

A <connect part> with at most one <state exit point> represents a *Connect-node*. If no <connect list> is given, the *State-exit-point-name* is omitted.

The <connect list> must only refer to visible <state exit point>s.

Semantics

A <connect part> represents an exit point on a <composite state>. Interpretation is resumed at this point if in the <composite state> there is interpretation of a <return> addressing a <state exit point> found in the <connect list>.

A <connect part> with an empty <connect list> corresponds to an unlabelled <return>, that is, <return> with no <expression>, in a <composite state>.

Model

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

A <connect list> that contains an <asterisk connect list> is transformed into a list of <state exit point>s, one for each <state exit point> of the <composite state> in question. The list of <state exit point>s is then transformed as described above.

11.12 Transition

11.12.1 Transition body

Abstract grammar

```
Transition          ::   Graph-node*
                         ( Terminator | Decision-node )
Graph-node          ::   ( Task-node
                          | Output-node
                          | Create-request-node
                          | Call-node
                          | Compound-node
                          | Set-node
                          | Reset-node ) [On-exception]
Terminator          ::   ( Nextstate-node
                          | Stop-node
                          | Return-node
                          | Join-node
                          | Continue-node
                          | Break-node
                          | Raise-node ) [On-exception]
```

Concrete textual grammar

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

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

<action statement> ::=
                        [<label>]
                        { <action 1> <end> [ <on exception> ] | <action 2> <end> }

<action 1> ::=
                        <task>
                  |     <output>
                  |     <create request>
                  |     <decision>
                  |     <set>
                  |     <reset>
                  |     <export>
                  |     <procedure call>
                  |     <remote procedure call>

<action 2> ::=
                        <transition option>

<terminator statement> ::=
                        [<label>]
                        { <terminator 1> <end> [ <on exception> ] | <terminator 2> <end> }

<terminator 1> ::=
                        <return>
                  |     <raise>

<terminator 2> ::=
                        <nextstate>
                  |     <join>
                  |     <stop>
```

If the <terminator statement> of a <transition> is omitted, then the last action in the <transition> must contain a terminating <decision> (see 11.13.5) 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>
                        |       <raise area> }

<transition string area> ::=
                        {       <task area>
                        |       <output 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 agent.

The <transition area> represents *Transition* and <transition string area> represents the list of *Graph-node*s.

Semantics

A transition performs a sequence of actions. During a transition, the data of an agent may be manipulated and signals may be output. The transition will end with the state machine of the agent entering a state, with a stop, with a return or with the transfer of control to another transition.

A transition in one process of a block can be interpreted at the same time as a transition in another process of the same block (provided they are not both enclosed by a process) or of another block. Transitions of processes contained in a process are interpreted interleaving, that is, only one contained process interprets a transition at a time until it reaches a nextstate (run-to-completion). A valid model of the interpretation of an SDL system is a complete interleaving of different processes at the level of all actions that cannot be transformed (by the rules in Annex F) into other actions, and are not excluded because they are in a transition alternating with a transition that is being interpreted (see 9.3).

An undefined amount of time may pass while an action is interpreted. It is valid for the time taken to vary each time the action is interpreted. It is also valid for the time taken to be the same at each interpretation or for it to be zero (that is the result of **now**, see 12.3.4.1, is not changed).

Model

A transition action may be transformed to a list of actions (possibly containing implicit states) according to the transformation rules for <import expression> and <remote procedure call>. To preserve an exception handler associated with the original action, terminator, or decision, this list of actions is encapsulated in a new, implicitly defined procedure with an anonymous name, here referred to as *Actions*, as follows, where list-of-actions refers to the resultant list of actions:

```
procedure Actions;
    start;
    list-of-actions;
    return;
endprocedure;
```

The old action is replaced by a call to *Actions*. If an exception handler was associated with the original action, the exception handler is associated with the call to *Actions*.

If the transformed construct occurred in a terminator or decision, the original terminator or decision is replaced by a call to *Actions*, followed by the new terminator or decision. If an exception handler was associated with the original terminator or decision, the exception handler is associated with the call to *Actions* and with the new terminator or decision.

No exception handler is associated with the body of *Actions* or with any part of this body.

11.12.2 Transition terminator

11.12.2.1 Nextstate

Abstract grammar

Nextstate-node	::	*State-name*
		[*Nextstate-parameters*]
Nextstate-parameters	::	[*Expression*]*
		[*State-entry-point-name*]
		[**HISTORY**]

Nextstate-parameters may only be present if *State-name* denotes a composite state.

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

Concrete textual grammar

```
<nextstate> ::=
                    nextstate <nextstate body>

<nextstate body>::=
                    <state name> [<actual parameters>] [ via <state entry point name> ]
                |   <dash nextstate>

<dash nextstate> ::=
                    <hyphen>
                |   <history dash nextstate>

<history dash nextstate> ::=
                    <history dash sign>
```

A *Nextstate-node* with **HISTORY** represents a <history dash nextstate>.

If a transition is terminated by a <history dash nextstate>, the <state> must be a <composite state>.

The <transition> contained in a <start> must not lead, directly or indirectly, to a <dash nextstate>. The <transition> contained in a <start> or a <handle> must not lead, directly or indirectly, to a <history dash nextstate>.

An <on exception> within a <start> or associated to a whole body must not, directly or indirectly (through <on exception>s within <exception handler>s), lead to an <exception handler> containing <dash nextstate>s.

Concrete graphical grammar

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

If <state entry point name> is given, the <nextstate> must refer to a composite state with the state entry point.

If <actual parameters> is given, the <nextstate> must refer to a composite state with <agent formal parameters>.

Semantics

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

A dash nextstate for a composite state implies that the next state is the composite state.

If a *State-entry-point-name* is given, the next state is a composite state, and interpretation continues with the *State-start-node* that has the same name in the *Composite-state-graph*.

When a *Nextstate-node* with **HISTORY** is interpreted, the next state is the one in which the current transition was activated. If interpretation re-enters a composite state, its entry procedure is invoked.

Model

In each <nextstate> of a <state> the <dash nextstate> is replaced by the <state name> of the <state>. This model is applied after the transformation of <state>s and all other transformations except those for trailing commas, synonyms, priority inputs, continuous signals, enabling conditions, implicit tasks for imperative actions and remote variables or procedures.

The rest of this *Model* section describes how the meaning of <dash nextstate> in exception handlers is determined.

An exception handler is called reachable from a state or exception handler if it is either associated to the state or exception handler, the stimuli attached to the state or exception handler or if it is associated with the transition actions following the stimuli. All exception handlers reachable from an exception handler that is reachable from the state are also called reachable from the state.

NOTE – Reachability is transitive.

For each <state>, the following rule applies: All reachable exception handlers are made distinct for the state by copying each exception handler to an <exception handler> with a new name. The <on exception>s are modified using this new name. Afterwards, exception handlers not reachable from any state are removed.

After this replacement, a given <exception handler> containing <dash nextstate>s can be reached, directly or indirectly, from exactly one <state>. The <dash nextstate>s within each such <exception handler> are replaced by the <state name> of this <state>.

11.12.2.2 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>.

Abstract grammar

Join-node :: *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 <agent type body> is stated in 8.3.1.

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 <state machine graph area>, there must be exactly one <in connector area> in that <state machine graph area>.

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

When a *Join-node* is interpreted, interpretation continues with the *Free-action* named with *Connector-name*.

11.12.2.3 Stop

Abstract grammar

Stop-node	::	( )

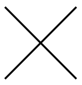
A *Stop-node* must not be contained in a *Procedure-graph*.

Concrete textual grammar

```
<stop> ::=
                              stop
```

Concrete graphical grammar

<stop symbol> ::=



Semantics

The stop causes the agent interpreting it to perform a stop.

This means that the retained signals in the input port are discarded and the state machine of the agent goes into a stopping state. When all contained agents have ceased to exist, the agent itself will cease to exist.

11.12.2.4 Return

Abstract grammar

Return-node	=	*Action-return-node*
	|	*Value-return-node*
	|	*Named-return-node*
Action-return-node	::	( )
Value-return-node	::	*Expression*
Named-return-node	::	*State-exit-point-name*

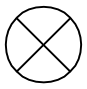
A *Return-node* must be contained in a *Procedure-graph* or *Composite-state-graph*. An *Action-return-node* must only be contained in the *Procedure-Graph* of a *Procedure-definition* without *Result* or *Composite-state-graph*. A *Value-return-node* must only be contained in the *Procedure-Graph* of a *Procedure-definition* containing *Result*. A *Named-return-node* must only be contained in a *Composite-state-graph*.

Concrete textual grammar

```
<return> ::=
                              return [<expression> | {via <state exit point>}]
```

Concrete graphical grammar

```
<return area> ::=
                              <return symbol>
                              [ is connected to <on exception association area> ]
                              [ is associated with {<expression> | <state exit point>} ]

<return symbol> ::=
```



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

<state exit point> is allowed if and only if the enclosing scope is a composite state containing the specified <state exit point>.

Semantics

A *Return-node* in a procedure 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) If a *Value-return-node* is interpreted, the result of *Expression* is returned to the calling context.

d) Hereafter, interpretation of the calling context continues at the node following the call.

A *Return-node* in a composite state results in activation of a *Connect-node*. For a *Named-return-node*, interpretation continues at the *Connect-node* with the same name. For an *Action-return-node*, interpretation continues at the *Connect-node* without a name.

11.12.2.5 Raise

Abstract grammar

Raise-node	::	*Exception-identifier*
		[*Expression*]*

The length of the list of optional *Expression*s must be the same as the number of *Sort-reference-identifiers* in the Exception-definition denoted by the *Exception-identifier*.

Each *Expression* must have a sort that is compatible to the corresponding (by position) *Sort-reference-identifier* in the *Exception-definition*.

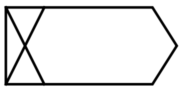
Concrete textual grammar

```
<raise> ::=
                    raise <raise body>

<raise body> ::=
                    <exception raise>

<exception raise> ::=
                    <exception identifier> [<actual parameters>]
```

A <raise> represents a *Raise-node*.

Concrete graphical grammar

```
<raise area> ::=
                    <raise symbol> contains <raise body>

<raise symbol> ::=
```



Semantics

Interpretation of a *Raise-node* creates an exception instance (see 11.16 for the interpretation of an exception instance). The data items that are conveyed by the exception instance are the results of the actual parameters of the <raise>. If an *Expression* in the list of optional *Expression*s is omitted (that is, if the corresponding <expression> in <actual parameters> is omitted), no data item is conveyed with the corresponding place of the exception instance, that is, the corresponding place is "undefined".

If a syntype is specified in the exception definition, and an expression is specified in the <raise>, the range check defined in 12.1.9.5 is applied to the expression.

Model

A raise may be transformed to a list of actions (possibly containing implicit states) plus a new raise according to the model (of remote procedure calls, for example). Then the model for transition terminators in 11.12.1 applies.

11.13 Action

11.13.1 Task

Abstract grammar

Task-node = *Assignment*
| *Assignment-attempt*
| *Informal-text*

Concrete textual grammar

<task> ::=
    task <textual task body>

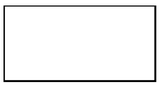
<textual task body> ::=
    <assignment>
  | <informal text>
  | <compound statement>

Concrete graphical grammar

<task area> ::=
    <task symbol> ***contains*** <graphical task body>
    [***is connected to*** <on exception association area>]

<graphical task body> ::=
    <statement list>
  | <informal text>

<task symbol> ::=



The trailing <end> in <statement list> of a <graphical task body> may be omitted.

Semantics

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

A task area creates its own scope.

Model

If the <statement list> in the <compound statement> of <textual task body> is empty, then the <task> is removed. If the <statement list> in a <graphical task body> is empty, the <task area> is removed. Any syntactic item leading to the <task> or <task area> shall then lead directly to the item following the <task> or <task area>, respectively.

A <task> containing a <compound statement> is transformed as shown in 11.14.1. The result of this transformation is inserted in place of <task>.

A <task area> containing a <statement list> is transformed as a <compound statement> in the same way as the <compound statement> in a <task> (see above), except that the result of transforming the <compound statement> in 11.14.1 is then transformed into its graphical counter part which replaces the <task area>.

NOTE – The transform of a <task> or <task area> containing a <statement list> is not necessarily mapped onto a *Task-node* in the Abstract Grammar.

11.13.2 Create

Abstract grammar

Create-request-node	::	[*Variable-identifier*]
		Agent-identifier
		[*Expression*]*

The length of the list of optional *Expression*s must be the same as the number of *Agent-formal-parameter*s in the *Agent-definition* of the *Agent-identifier*.

Each *Expression* corresponding by position to an *Agent-formal-parameter* must have a sort that is compatible to the sort of the *Agent-formal-parameter* in the *Agent-definition* denoted by *Agent-identifier*.

Concrete textual grammar

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

<create body> ::=
    { <agent identifier> | <agent type identifier> | **this** } [<actual parameters>]

<actual parameters> ::=
    (<actual parameter list> **)**

<actual parameter list> ::=
    [<expression>] { **,** [<expression>] }*

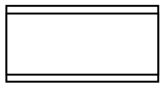
Commas after the last <expression> in <actual parameter list> may be omitted.

this may only be specified in an <agent type definition> and in scopes enclosed by an <agent type definition>.

Concrete graphical grammar

<create request area> ::=
    <create request symbol> ***contains*** <create body>
    [***is connected to*** <on exception association area>]

<create request symbol> ::=



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

Semantics

The create action causes the creation of an agent instance either inside the agent that performs the create or in the agent that contains the agent that performs the create. The parent of the created agents (see clause 9, *Model*) has the same pid as returned by **self** of the creating agent. **self** of the created agents (see clause 9, *Model*) and offspring of the creating agent (see clause 9, *Model*) both have the same unique, new pid.

When an agent 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 12.3.3) to the corresponding formal parameters. If the created agent has contained agent sets, then the initial instances of these sets are created. Then the agent starts by interpreting the start node in the agent graph and the start nodes of the initial contained agents are interpreted in some order, before transitions caused by signals are interpreted.

The created agent is then interpreted asynchronously and concurrently or alternating with other agents depending on the kind of the containing agent (system, block, process).

If an attempt is made to create more agent instances than specified by the maximum number of instances in the agent definition, then no new instance is created, the offspring of the creating agent (see clause 9, *Model*) has the result Null and interpretation continues.

If an <expression> in <actual parameters> is omitted, the corresponding formal parameter has no data item associated; that is, it is "undefined".

If the <agent type identifier> is used in a <create request>, then the corresponding agent type may not be defined as <abstract> or contain formal context parameters.

If both an instance set and an agent type with the same name are defined in a scope unit and a create statement in this scope unit uses this name, then an instance is created in the instance set and not based on the agent type. Note that it is possible to create an instance of the agent type by defining an instance set based on the agent type and then creating an instance in this set.

Model

Stating **this** is derived syntax for the implicit <process identifier> that identifies the set of instances of the agent in which the create is being interpreted.

If <agent type identifier> is used in a <create request> the following models apply:

a) If there exists one instance set of the indicated agent type in the agent containing the instance that performs the create, the <agent type identifier> is derived syntax denoting this instance set.

b) If there is more than one instance set it is determined at interpretation time in which set the instance will be created. The <create request> is in this case replaced by a non-deterministic decision using **any** followed by one branch for each instance set. In each of the branches a create request for the corresponding instance set is inserted.

c) If there does not exist any instance set of the indicated agent type in the containing agent then:
   i) an implicit instance set of the given type with a unique name is created in the containing agent; and
   ii) the <agent identifier> in the <create request> is derived syntax for this implicit instance set.

11.13.3 Procedure call

Abstract grammar

Call-node	::	*Procedure-identifier*
		[*Expression*]*
Value-returning-call-node	::	*Procedure-identifier*
		[*Expression*]*

The length of the list of optional *Expression*s must be the same as the number of the *Procedure-formal-parameter*s in the *Procedure-definition* denoted by the *Procedure-identifier*.

Each *Expression* corresponding by position to an *In-parameter* must be sort compatible to the sort of the *Procedure-formal-parameter*.

Each *Expression* corresponding by position to an *Inout-parameter* or *Out-parameter* must be a *Variable-identifier* which is sort compatible to the sort identified by the *Sort-reference-identifier* of the *Procedure-formal-parameter*.

Concrete textual grammar

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

<procedure call body> ::=
    [**this**] { <procedure identifier> | <procedure type expression> } [<actual parameters>]

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

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

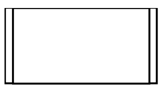
If **this** is used, <procedure identifier> must denote an enclosing procedure.

The <procedure call> represents a *Call-node*. A <value returning procedure call> (see 12.3.5) represents a *Value-returning-call-node*.

Concrete graphical grammar

<procedure call area> ::=
    <procedure call symbol> ***contains*** <procedure call body>
    [***is connected to*** <on exception association area>]

<procedure call symbol> ::=



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

Semantics

The interpretation of a procedure *Call-node* or *Value-returning-call-node* interprets the actual parameter expressions in the order given and then transfers the interpretation to the procedure definition referenced by the *Procedure-identifier*, and that procedure graph is interpreted (the explanation is contained in 9.4).

The interpretation of the transition containing a *Call-node* continues when the interpretation of the called procedure is finished.

The interpretation of the transition containing a *Value-returning-call-node* continues when the interpretation of the called procedure is finished. The result of the called procedure is returned by the *Value-returning-call-node*.

If an <expression> in <actual parameters> is omitted, the corresponding formal parameter has no data item associated; that is, it is "undefined".

Model

If the <procedure identifier> is not defined within the enclosing agent, 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.

11.13.4 Output

Abstract grammar

Output-node	::	*Signal-identifier* [*Expression*]* [*Signal-destination*] *Direct-via*
Signal-destination	=	*Expression* | *Agent-identifier*
Direct-via	=	(*Channel-identifier* | *Gate-identifier*)***-set***
Channel-identifier	=	*Identifier*

The length of the list of optional *Expression*s must be the same as the number of *Sort-reference-identifier*s in the *Signal-definition* denoted by the *Signal-identifier*.

Each *Expression* must be sort compatible to the corresponding (by position) *Sort-identifier-reference* in the *Signal-definition.*

For each *Channel-identifier* in *Direct-via* there must exist zero or more channels such that the channel via this path is reachable with the *Signal-identifier* from the agent, and the *Channel-path* in the direction from the agent must include *Signal-identifier* in its set of *Signal-identifier*s.

For each *Gate-identifier* in *Direct-via,* there must exist zero or more channels such that the gate via this path is reachable with the *Signal-identifier* from the agent and the *Out-signal-identifier-**set*** of the gate must include the *Signal-identifier*.

Concrete textual grammar

```
<output> ::=
                **output** <output body>

<output body> ::=
                <signal identifier> [<actual parameters>] {, <signal identifier> [<actual parameters>] }*
                <communication constraints>

<destination> ::=
                <pid expression> | <agent identifier> | this

<via path> ::=
                via { <channel identifier> | <gate identifier> }
```

The <pid expression> or the <agent identifier> in <destination> represents the *Signal-destination.* There is a syntactic ambiguity between <pid expression> and <agent 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 <agent identifier>. <agent identifier> must denote an agent, which is reachable from the originating agent.

The <communication constraints> (see 10.5) in an <output> shall contain no **timer <**timer identifier> clause. It contains at most one **to** <destination> clause and zero or more <via path>s.

Each <via path> of <communication constraints> represents a *Channel-identifier* or *Gate-identifier* in the *Direct-via*.

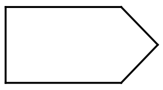
this may only be specified in an <agent type definition> and in scopes enclosed by an <agent type definition>.

If <destination> is a <pid expression> with a static sort other than Pid (see 12.1.6), the <signal identifier> must represent a signal defined or used by the interface that defined the pid sort.

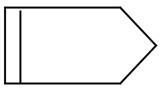
The <gate identifier> in <via path> may be used to identify a gate that is defined using <textual interface gate definition> or <graphical interface gate definition>.

Concrete graphical grammar

```
<output area> ::=
                    <output symbol> contains <output body>
                    [ is connected to <on exception association area> ]

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

<plain output symbol> ::=
```



```
<internal output symbol>::=
```



NOTE 1 – There is no difference in meaning between a <plain output symbol> and an <internal output symbol>.

Semantics

Stating an *Agent-identifier* in *Signal-destination* indicates *Signal-destination* as any existing instance of the set of agent instances indicated by *Agent-identifier*. If no instances exist, the signal is discarded.

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

If there is a process instance that contains both the sender and the receiver, then the data items conveyed by the signal instance are the results of the actual parameters of the output. Otherwise, the data items conveyed by the signal instance are newly created replicates of the results of the actual parameters of the output and share no references with the results of the actual parameters of the output. When there are cycles of references in the result of the actual parameters, the conveyed data items will also contain these cycles. Each conveyed data item will be equal to the corresponding actual parameter of the output.

If an <expression> in <actual parameters> is omitted, no data item is conveyed with the corresponding place of the signal instance, that is, the corresponding place is "undefined".

The pid of the originating agent 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 12.1.9.5 is applied to the expression.

If <destination> is a <pid expression> and the static sort of the pid expression is Pid, then the compatibility check for the dynamic sort of the pid expression (see 12.1.6) is performed for the signal denoted by the *Signal-identifier*.

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 path>s clause to include at least one of the paths mentioned in the *Direct-via*.

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

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

NOTE 2 – If *Signal-destination* is 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 an agent instance set, which can be reached by the communication paths able to convey the signal instance. This agent instance set is arbitrarily chosen. Second, when the signal instance arrives at the end of the communication path, it is delivered to an instance of the agent 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 *Gate-identifier* in the *Direct-via* of two *Output-node*s does not automatically mean that the signals are queued in the input port in the same order as the *Output-node*s are interpreted. However, order is preserved if the two signals are conveyed by identical delaying channels, or only conveyed by channels with no delay.

Model

If several pairs of <signal identifier> and <actual parameters> are specified in an <output body>, this is derived syntax for specifying a sequence of <output>s or <output area>s in the same order as specified in the original <output body>, each containing a single pair of <signal identifier> and <actual parameters>. The **to** <destination> clause and the <via path>s are repeated in each of the <output>s or <output area>s.

Stating **this** in <destination> is derived syntax for the implicit <agent identifier> that identifies the set of instances for the agent in which the output is being interpreted.

11.13.5 Decision

Abstract grammar

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

The *Range-condition*s of the *Decision-answer*s must be mutually exclusive, and the *Constant-expression*s of the *Range-condition*s must be of a compatible sort. If the *Decision-question* is an *Expression*, the *Range-condition* of the *Decision-answer*s must be sort compatible to the sort of the *Decision-question*.

Concrete textual grammar

```
<decision> ::=
                    decision <question> <end> [<on exception>]
                        <decision body>
                    enddecision

<decision body> ::=
                    <answer part>+ [<else part>]

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

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

<else part> ::=
                    else <colon> [<transition>]
```

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

An <answer part> or <else part> in a decision or a transition option is a terminating <answer part> or <else part> respectively if it contains a <transition> where a <terminator statement> is specified, or contains a <transition string> whose last <action statement> contains a terminating decision or option. A <decision> or <transition option> is a terminating decision and terminating transition option respectively, if each <answer part> and <else part> in its <decision body> is a terminating <answer part> or <else part> respectively.

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.

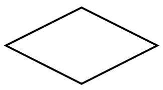
There is syntactic ambiguity between <informal text> and <character string> in <question> and <answer>. If the <question> and all <answer>s are <character string>s, all of these are interpreted as <informal text>. If the <question> or any <answer> is a <character string>, that does not match the context of the decision, the <character string> denotes <informal text>.

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

Concrete graphical grammar

<decision area> ::=
    <decision symbol> ***contains*** <question>
    [***is connected to*** <on exception association area>]
    is followed by <graphical decision body>

<decision symbol> ::=



<graphical decision body> ::=
    { <graphical answer part>+ [<graphical else part>] } ***set***

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

<graphical answer> ::=
    [<answer>] | **(** [<answer>] **)**

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

The <graphical answer> and **else** may be placed along the associated <flow line symbol>, or over the <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 result 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 result of the expression on which the question is posed is not covered by the results specified in the other answers.

Whenever the *Else-answer* is not specified, and the result from the evaluation of the question expression does not match one of the answers, then the predefined exception NoMatchingAnswer is raised.

Model

If a <decision> is not terminating, it is derived syntax for a <decision> where all not terminating <answer part>s and the <else part> (if not terminating) have inserted at the end of 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 shorthand for using <any 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.

11.14 Statement list

A statement list can be used in a <task>, <task area>, <procedure definition>, or <operation definition> to define variables local to the statement list and a number of actions to be interpreted. The purpose of a statement list is to allow concise textual descriptions of algorithms to be combined with either SDL/GR or the less concise SDL/PR form. The semantics of a statement list are determined by transformation of the statements according to the models below, so that the statements are effectively interpreted left to right.

A variable definition statement can introduce variables at the beginning of a <statement list>. In contrast to <variable definition> in 12.3.1, initialization of variables in this context is not required to be a <constant expression>.

Concrete textual grammar

```
<statement list> ::=
                        <variable definitions> <statements>

<variable definitions> ::=
                        { <variable definition statement> }*

<statements> ::=
                        <statement>*

<statement> ::=
                        <empty statement>
                   |    <compound statement>
                   |    <assignment statement>
                   |    <algorithm action statement>
                   |    <call statement>
                   |    <expression statement>
                   |    <if statement>
                   |    <decision statement>
                   |    <loop statement>
                   |    <terminating statement>
                   |    <labelled statement>
                   |    <exception statement>
```

<terminating statement> ::=
<return statement>
| <break statement>
| <loop break statement>
| <loop continue statement>
| <raise statement>

A <loop break statement> and <loop continue statement> may only occur within a <loop statement>.

A <terminating statement> may only occur as the last <statement> in <statements>. If the last <statement> in <statement list> is a <terminating statement>, the <statement list> is terminating.

<variable definition statement> ::=
dcl <local variables of sort> { **,** <local variables of sort> }* <end>

<local variables of sort> ::=
<variable name> { **,** <variable name>}* <sort> [<is assigned sign> <expression>]

11.14.1 Compound statement

Multiple statements may be grouped into a single statement.

Abstract grammar

Compound-node	::	*Connector-name*
		*Variable-definition-**set***
		*Init-graph-node**
		Transition
		*Step-graph-node**
Init-graph-node	=	*Graph-node*
Step-graph-node	=	*Graph-node*
Continue-node	::	*Connector-name*
Break-node	::	*Connector-name*

Concrete textual grammar

<compound statement> ::=
<left curly bracket> <statement list> <right curly bracket>

The <compound statement> represents a *Compound-node*. The *Connector-name* is represented by a newly created anonymous name. The *Variable-definition-**set*** is represented by the list of all <variable definition>s in <statement list>. The *Transition* is represented by the transform of <statements> in <statement list>, or by the transform of <statements> in <statement list> followed by a *Break-node* with *Connector-name*, if the <statement list> is not terminating.

Semantics

A <compound statement> creates its own scope.

The interpretation of a *Compound-node* proceeds as follows:

a) A local variable is created for each *Variable-definition* in the *Variable-definition-**set***.

b) The list of *Init-graph-node*s is interpreted.

c) The *Transition* is interpreted.

d) When a *Continue-node* with a *Connector-name* matching *Connector-name* is interpreted, the list of *Step-graph-node*s is interpreted and further interpretation continues at step (c).

e) When the interpretation of the *Compound-node* terminates, all variables created by the interpretation of the *Compound-node* will cease to exist. Interpretation of a *Compound-node* terminates:

   i) when a *Break-node* is interpreted; or

   ii) when a *Continue-node* with a *Connector-name* different from the *Connector-name* in *Compound-node* is interpreted; or

iii) when a *Return-node* is interpreted; or

iv) when an exception instance is created that is not handled within the *Transition* of the *Compound-node*.

f) Hereafter, interpretation continues as follows:

i) If the interpretation of the *Compound-node* terminated due to the interpretation of a *Break-node* with a *Connector-name* matching *Connector-name*, then interpretation continues at the node following the *Compound-node*; otherwise

ii) if the interpretation of the *Compound-node* terminated due to the interpretation of a *Break-node*, *Continue-node* or *Return-node*, then the interpretation continues with interpretation of the *Break-node*, *Continue-node* or *Return-node*, respectively, at the point of invocation of the *Compound-node*; otherwise

iii) if the interpretation of the *Compound-node* terminated due to the creation of an exception instance the interpretation continues as described in 11.16.

Model

If the <statement list> contains <variable definitions>, the following is performed for each <variable definition statement>. A new <variable name> is created for each <variable name> in the <variable definition statement>. Each occurrence of <variable name> in the following <variable definition statement>s and within <statements> is replaced by the corresponding newly created <variable name>.

For each <variable definition statement>, a <variable definition> is formed from the <variable definition statement> by omitting the initializing <expression> (if present) and inserted as a <variable definition statement> in place of the original <variable definition statement>. If an initializing <expression> is present, an <assignment statement> is constructed for each <variable name> mentioned in the <local variables of sort> in the order of their occurrence, where <variable name> is given the result of <expression>. These <assignment statement>s are inserted at the front of <statement>s in the order of their occurrence.

The <statement list> is equivalent to the concatenation of the transform of each <variable definition statement> and the transform of each <statement> in <statements> (see 11.14.1 to 11.14.7).

NOTE – The transformed non-empty <statement list> becomes a list of <action statement>s and <terminator statement>s separated by semicolons and ending in a semicolon and therefore can be treated as a <transition>.

If the <statement list> is empty, the result of its transformation is the empty text.

11.14.2 Transition actions and terminators as statements

Within a statement list, an assignment statement is not preceded by the **task** keyword, and a procedure call does not need the **call** keyword. Most of the other <action statement>s that can be used in a transition (see 11.12.1) and the <return> terminator of a transition (see 11.12.1) can be used as <statement>s in a <statement list>.

Concrete textual grammar

```
<assignment statement> ::=
                              <assignment> <end>

<algorithm action statement> ::=
                              <output> <end>
                         |    <create request> <end>
                         |    <set> <end>
                         |    <reset> <end>
                         |    <export> <end>

<return statement> ::=
                              <return> <end>
```

<raise statement> ::=

    <raise> <end>

<call statement> ::=

    [**call**] <procedure call body> <end>

A <return statement> is only allowed within a <procedure definition> or within an <operation definition>.

The keyword **call** cannot be omitted if the <call statement> is syntactically ambiguous with an operation application or variable with the same name.

NOTE – This ambiguity is not resolved by context.

Model

<assignment statement> is transformed into the <task>:

```
task <assignment> ;
```

A <call statement> is derived syntax for <procedure call> and is transformed into a <procedure call> with the same <procedure call body>:

```
call <procedure call body> ;
```

The transform of an <algorithm action statement>, a <return statement>, and <raise statement> is obtained by dropping the trailing <end>.

11.14.3 Expressions as Statements

Expressions that are operation applications can be used as statements, in which case the <operation application> is interpreted and the result is ignored.

Concrete textual grammar

<expression statement> ::=

    <operation application> <end>

Model

A new <variable name> is created. A <variable definition> is added at the end of *Decl-list* that declares the newly created <variable name> to be of the same sort as the result of <operation application>. Finally, the expression statement is transformed to:

```
task <variable name> := <operation application> ;
```

11.14.4 If statement

The <Boolean expression> is interpreted and if it returns the predefined Boolean value true, the <consequence statement> is interpreted, otherwise the <alternative statement>, if present, is interpreted.

Concrete textual grammar

<if statement> ::=

    if (<Boolean expression> **)** <consequence statement>
        [**else** <alternative statement>]

<consequence statement> ::=

    <statement>

<alternative statement> ::=

    <statement>

An <alternative statement> associates with the closest preceding <consequence statement>.

Model

The <if statement> is equivalent to the following <action statement> involving a <decision>:

```
decision <Boolean expression> ;
   ( true ) : task { <consequence statement>-transform };
   ( false ) : task { <alternative statement>-transform };
 enddecision ;
```

The transform of <alternative statement> is only inserted if <alternative statement> was present.

11.14.5 Decision statement

The decision statement is a concise form of decision. The <expression> is evaluated and the <algorithm answer part> whose <range condition> contains the result of the expression is interpreted. Overlapping range conditions are not allowed. Unlike in a <decision> (see 11.13.5), it is not necessary for the expression to match one of the range conditions. If there is no match and an <alternative statement> exists, the <alternative statement> is interpreted. If there is no match and an <alternative statement> does not exist, interpretation continues after the <decision statement>.

Concrete textual grammar

```
<decision statement> ::=
                         decision ( <expression> ) <left curly bracket>
                               <decision statement body>
                         <right curly bracket>

<decision statement body> ::=
                          <algorithm answer part>+ [<algorithm else part>]

<algorithm answer part> ::=
                         ( <range condition> ) <colon> <statement>

<algorithm else part> ::=
                         else <colon> <alternative statement>
```

Model

An <algorithm answer part> is transformed into the following <answer part>.

```
( <range condition> ) : task { <statement>-transform };
```

A <decision body> is then formed by taking the transform of each <algorithm answer part> in order and appending the following <else part>.

```
else : task { <alternative statement>-transform };
```

where the transformation of <alternative statement> is only inserted if <alternative statement> is present. The resulting <decision body> is referred to as *Body*. The <decision statement> is equivalent to the following <action statement>:

```
decision ( <expression> ) ;
   Body
enddecision ;
```

11.14.6 Loop statement

The <loop statement> provides a generalized facility for bounded or unbounded iteration of a <loop body statement>, with an arbitrary number of loop variables. These variables may be defined within the <loop statement> and are stepped in ways specified by the <loop step>. They may be used both to generate successive results and to accumulate results. When the <loop statement> terminates, a <finalization statement> may be interpreted in the context of the loop variables.

The <loop body statement> is interpreted repeatedly. The interpretation of the loop is controlled by the presence of any <loop clause>. A <loop clause> may begin with a <loop variable indication>

which provides a convenient way to declare and initialize local loop variables. The scope and lifetime of any variable defined in a <loop variable indication> are effectively those of the <loop statement>. If initialization is present as an <expression> in a <loop variable definition>, the expression is evaluated only once before the first interpretation of the loop body. Alternatively, any visible variable can be defined as a loop variable and can have a data item assigned to it. Before each iteration, all <Boolean expression> elements are evaluated. Interpretation of the <loop statement> is terminated if any one <Boolean expression> element returns false. Consequentially, if there is no <Boolean expression> present, interpretation of the <loop statement> will continue until the <loop statement> is exited non-locally. If a <loop variable indication> is present in that <loop clause>, the <loop step> in each loop clause computes and assigns the result of the respective loop variable at the end of each iteration. If a <loop variable indication> was not present in a <loop clause>, or if a <loop step> was not present, no assignment statement to the loop variable is performed. The <loop variable indication>, <Boolean expression>, and <loop step> are optional. A loop variable is visible but must not be assigned to in the <loop body statement>.

Interpretation of the loop body also terminates when a **break** is reached. Reaching a **continue** statement causes interpretation of the loop to jump immediately to the next iteration. (See also <break statement> in 11.14.7).

If a <loop statement> is terminated "normally" (that is by a <Boolean expression> evaluating to the predefined Boolean value false) the <finalization statement> is interpreted. A loop variable is visible and retains its result when the <finalization statement> is interpreted. A break or continue statement within the <finalization statement> terminates the next outer <loop statement>.

Concrete textual grammar

```
<loop statement> ::=
                         for ( [ <loop clause> { ; <loop clause> }* ] )
                              <loop body statement> [ then <finalization statement> ]

<loop body statement> ::=
                         <statement>

<finalization statement> ::=
                         <statement>

<loop clause> ::=
                         [<loop variable indication>]
                         , [<Boolean expression>]
                         <loop step>

<loop step> ::=
                         [ , [ { <expression> | [ call ] <procedure call body> } ] ]

<loop variable indication> ::=
                         <loop variable definition>
                    |    <variable identifier> [ <is assigned sign> <expression> ]

<loop variable definition> ::=
                         dcl <variable name> <sort> <is assigned sign> <expression>

<loop break statement> ::=
                         break <end>

<loop continue statement> ::=
                         continue <end>
```

The keyword **call** cannot be omitted in a <loop step> if this would lead to an ambiguity with an operation application or variable with the same name.

The <procedure identifier> in the <procedure call body> of a <loop step> must not refer to a value returning procedure call.

A <finalization statement> associates with the closest preceding <loop body statement>.

A <loop statement> represents a *Compound-node*. The *Connector-name* is represented by a newly created anonymous name, referred to as *Label*.

The *Variable-definition-**set*** is represented by the list of <variable definition statement>s constructed from the <variable name> and <sort> mentioned in each <loop variable definition>.

The list of *Init-graph-node*s is represented by the transform of the <statement list> constructed from <assignment statement>s formed from each <loop variable indication> in the order of their occurrence.

An <assignment statement> is constructed from each <loop clause> between the <variable name> or <variable identifier> and the <expression> in <loop step>, if both <loop variable indication> and <expression> were present. <statements> is constructed by taking these <assignment statement> elements in sequence, or the <expression>, <procedure call>, or <procedure call body> in <loop step>, if no <assignment statement> was constructed. <statements> represents the list of *Step-graph-node*s.

The *Transition* is represented by the following <decision>, where *Limit* refers to the <expression> constructed by combining all <Boolean expression> items through the predefined operator "and" of type Boolean:

```
decision Limit ;
( True ) : task { <loop body statement>-transform }; Continue ;
( False ) : task { <finalization statement>-transform };
enddecision ;
```

The transform of <finalization statement> is inserted only if a <finalization statement> was originally present.

A <loop continue statement> represents a *Continue-node*. The *Connector-name* is represented by *Label* of the innermost enclosing loop statement.

Model

Every occurrence of a <loop break statement> inside a <loop clause> or the <loop body statement> or a <finalization statement> of another <loop statement> contained within this <loop statement>, all not occurring within another inner <loop statement>, is replaced by:

```
break Label ;
```

If a <Boolean expression> is absent in a <loop clause>, the predefined Boolean value true is inserted as the <Boolean expression>.

Then the <loop statement> is replaced by the so modified <loop statement> followed by a <labelled statement> with <connector name> *Break*.

11.14.7 Break and labelled statements

A <break statement> is a more restrictive form of a <join>.

A <break statement> causes the interpretation to be immediately transferred to the statement following the one with the matching <label>.

Concrete textual grammar

```
<break statement> ::=
                    break <connector name> <end>

<labelled statement> ::=
                    <label> <statement>
```

A <break statement> must be contained in a statement that has been labelled with the given <connector name>.

A <break statement> represents a *Break-node* with the *Connector-name* represented by <connector name>.

A <labelled statement> represents a *Compound-node* with the *Connector-name* represented by the <connector name> in <label>.

11.14.8 Empty Statement

A statement may be empty, signified by using a single semicolon. The <empty statement> has no effect.

Concrete textual grammar

```
<empty statement> ::=
                    <end>
```

Model

The transform of the <empty statement> is the empty text.

11.14.9 Exception Statement

A statement can be encapsulated within an exception handler.

Concrete textual grammar

```
<exception statement> ::=
                    try <try statement> <handle statement>+
<try statement> ::=
                    <statement>
<handle statement> ::=
                    handle ( <exception stimulus list> ) <statement>
```

A <handle statement> associates with the closest preceding <try statement>. The <try statement> must not be a <break statement>.

The exception handler constructed in *Model* represents the *On-exception* of the *Local-scope-node* represented by the <compound statement> that is obtained from the <try statement> (see 11.14.1).

Semantics

An <exception statement> creates its own scope with an exception handler.

Model

If the <try statement> was not a <compound statement>, the <try statement> is first transformed into a <compound statement>:

```
{ <try statement> }
```

Then the (transformed) <try statement> and all <handle statement>s are transformed. For each <handle statement>, the following <handle> is constructed:

```
handle <exception stimulus list> ;
        task { <handle statement>-transform };
```

The constructed <handle>s are collected into a list referred to as *HandleParts*. An exception handler is constructed having an anonymous name. The name of the exception handler is referred to as *handler*.

```
exceptionhandler handler;
    HandleParts
endexceptionhandler;
```

The transform of the <exception statement> is the transform of the <try statement>.

11.15 Timer

Abstract grammar

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

The sorts of the list of *Expression*s in the *Set-node* and *Reset-node* must correspond by position to the list of *Sort-reference-identifier*s 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>] [<timer default initialization>]

<timer default initialization> ::=
　　<is assigned sign> <Duration constant expression>

<reset> ::=
　　reset (<reset clause> { **,** <reset clause> }* **)**

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

<set> ::=
　　set <set clause> { **,** <set clause> }*

<set clause> ::=
　　([<Time expression> **,**] <timer identifier> [**(**<expression list>**)**] **)**

A <set clause> may omit <Time expression>, if <timer identifier> denotes a timer which has a <timer default initialization> in its definition.

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

There is no corresponding graphical syntax for <reset> and <set>.

Semantics

A timer instance is an object, which 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 equality expression (see 12.2.7) applied to all corresponding expressions in the two lists yields the predefined Boolean value true (that is, if the two expression lists have the same result).

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 agent. 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 result as the **self** expression. If an expression list is given when the timer is set, the results 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 12.1.9.5 applied to the corresponding expression in a set or reset must be the predefined Boolean value true, otherwise the predefined exception OutOfRange is raised.

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 clause> with no <Time expression> is derived syntax for a <set clause> where <Time expression> is:

now + <Duration constant expression>

where <Duration constant expression> is derived from the <timer default initialization> in timer definition.

A <reset> or a <set> may contain several <reset clause>s or <set clause>s respectively. This is derived syntax for specifying a sequence of <reset>s or <set>s, one for each <reset clause> or <set clause> 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.

11.16 Exception

An exception instance transfers control to an exception handler.

Abstract grammar

Exception-definition	::	*Exception-name*
		*Sort-reference-identifier**
Exception-name	=	*Name*
Exception-identifier	=	*Identifier*

Concrete textual grammar

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

<exception definition item> ::=
    <exception name> [<sort list>]

Semantics

An exception instance denotes that an exceptional situation (typically an error situation) has occurred while interpreting a system. An exception instance is created implicitly by the underlying system or explicitly by a *Raise-node* and the exception instance ceases to exist if it is caught by a *Handle-node* or *Else-handle-node*.

Creation of an exception instance breaks the normal flow of control within an agent, operation or procedure. If an exception instance is created within a called procedure, operation, or compound statement and is not caught there, the procedure, operation, or compound statement, respectively, terminates and the exception instance propagates (dynamically) outwards to the caller and is treated as if it were created at the place of the procedure call, operation application, or invocation of the compound statement. This rule also holds for calls of remote procedures; and in this case, the exception instance propagates back to the calling process instance in addition to being propagated within the called agent instance.

A number of exception types are predefined within the package Predefined. These exception types are the ones that can be created by the underlying system implicitly. It is also allowed for the specifier to create instances of these exception types explicitly.

If an exception instance is created within an agent instance and is not caught there, the further behaviour of the system is undefined.

11.16.1 Exception handler

Abstract grammar

Exception-handler-node	::	*Exception-handler-name*
		[*On-exception*]
		*Handle-node-**set***
		[*Else-handle-node*]
Exception-handler-name	=	*Name*

The Exception-handler-nodes within a given *State-transition-graph* or *Procedure-graph* must all have different Exception-handler-names.

NOTE – An *Exception-handler-name* can have the same name as a *State-name*. They are however different.

The *Exception-identifier*s in the *Handler-node-**set*** must be distinct.

Concrete textual grammar

```
<exception handler> ::=
                    exceptionhandler <exception handler list> <end>
                           [<on exception>]
                           <handle>*
                    [ endexceptionhandler [ <exception handler name> ] <end> ]

<exception handler list> ::=
                    <exception handler name> { , <exception handler name> }*
              |     <asterisk exception handler list>

<asterisk exception handler list> ::=
                    <asterisk> [ ( <exception handler name> { , <exception handler name>}* ) ]
```

When the <exception handler list> contains one <exception handler name>, the <exception handler name> represents an *Exception-handler-node*. For each *Exception-handler-node*, the *Handle-node-**set*** is represented by the <handle>s containing <exception identifier>s in their <exception stimulus list>s. For each *Exception-handler-node*, the *Else-handle-node* is represented by an explicit or implicit <handle> in which the <exception stimulus list> is an <asterisk exception stimulus list>.

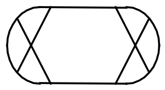
The <exception handler name>s in an <asterisk exception handler list> must be distinct and must be contained in other <exception handler list>s in the enclosing body or in the body of a supertype.

An <exception handler> contains at most one <asterisk exception stimulus list> (see 11.16.3).

An <exception handler> has at most one <exception handler> associated.

Concrete graphical grammar

```
<exception handler area> ::=
                    <exception handler symbol> contains <exception handler list>
                    [ is connected to <on exception association area>]
                    is associated with <exception handler body area>

<exception handler symbol> ::=
```



<exception handler body area> ::=
    <handle association area>*

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

An <exception handler area> represents one or more *Exception-handler-node*s. The <solid association symbol>s originating from an <exception handler symbol> may have a common originating path. An <exception handler area> must contain <state name> (not <asterisk state list>) if it coincides with an <on exception area>.

Semantics

An exception handler represents a particular condition in which an agent, operation, or procedure may handle an exception instance that it has created. Handling an exception instance results in a transition. The state of the process or procedure is not changed.

If the *Exception-handler-node* has no *Handle-node* with the same *Exception-identifier* as the exception instance, the exception instance is caught by the *Else-handle-node*. If there is no *Else-handle-node*, the exception instance is not handled in that exception handler.

Model

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

An <exception handler> with an <asterisk exception handler list> is transformed to a list of <exception handler>s, one for each <exception handler name> of the body in question, except for those <exception handler name>s contained in the <asterisk exception handler list>.

11.16.2 On-Exception

Abstract grammar

On-exception :: *Exception-handler-name*

The *Exception-handler-name* specified in *On-exception* must be the name of an <exception handler> within the same *State-transition-graph* or *Procedure-graph*.

Concrete textual grammar

<on exception> ::=
    onexception <exception handler name> <end>

An <exception handler name> may appear in more than one <exception handler> of a body.

Concrete graphical grammar

<on exception association area> ::=
    <solid on exception association symbol> ***is connected to***
        { <on exception area> | <exception handler area> }

<solid on exception association symbol> ::=
    $\longrightarrow$

<on exception area> ::=
    <exception handler symbol> ***contains*** <exception handler name>

A <solid on exception association symbol> may consist of several horizontal and vertical line segments. The arrowhead must be attached to the <on exception area> or <exception handler area>.

Semantics

An *On-exception* indicates which exception handler an agent, operation, or procedure should enter if the agent or procedure creates an exception instance. Through an <on exception> or <on exception association area> an exception handler is associated to another entity. An exception handler is said to be active whenever it is able to react on creation of an exception instance.

Several exception handlers may be active at the same time. For each agent, procedure or operation instance, there are several exception scopes that might contain an active exception handler. The exception scopes, in the order of increasing locality, are:

a) the entire graph of the instance;

b) the composite states (if a composite state is being interpreted);

c) the graph of the composite states (if any);

d) the current state;

e) the transition for the stimulus in the current state, or the start transition;

f) the current exception state;

g) the transition for the current handle clause; and

h) the current action.

Due to nesting of composite states, more than one exception handler for a composite state or composite state graph may be active at any time.

When an exception instance is created, the active exception handlers are visited in the order of decreasing locality. When an exception state is visited, the exception handler is of the current exception scope deactivated. If no exception handler is active for a certain exception scope, or if the exception state does handle the exception, the next exception scope is visited.

No exception handler is active during the interpretation of a <constant expression>.

An exception handler may be associated to a whole agent/procedure/operation graph, a start transition, a state, an exception handler, a state trigger (e.g. input or handle) with its associated transition, a transition action (most kinds of), or a transition terminator (some kinds of). The following text describes for each case when the exception handler is activated and deactivated.

a) *Whole agent/procedure/operation graph*

   The exception handler is activated at the start of interpretation of the graph of the agent, operation or procedure instance; the exception handler is deactivated when the agent, operation or procedure instance is in the stopping condition or ceases to exist.

b) *Start transition*

   The exception handler is activated when interpretation of the start transition starts in the agent, operation or procedure; the exception handler is deactivated when the agent or procedure interprets a nextstate node or is in the stopping condition or ceases to exist.

c) *Composite State*

   The exception handler is activated when the composite state is entered; it is active for the composite state including any *Connect-nodes* or transitions attached to the state. It is deactivated when interpretation enters another state.

d) *Composite State Graph*

   The exception handler is activated before the entry procedure of a composite state is invoked. It is deactivated after the exit procedure of the composite state is completed.

e) State

The exception handler is activated whenever the agent or procedure enters the given state. The exception handler is deactivated when the agent or procedure interprets a nextstate node or enters a stopping condition or ceases to exist.

f) *Exception handler*

The exception handler is activated whenever the agent or procedure enters the given exception handler; the exception handler is deactivated when the agent or procedure enters a nextstate node or enters a stopping condition or ceases to exist.

g) *Input*

The exception handler for the stimulus is activated whenever interpretation of the given Input-node is started in the agent or procedure. The exception handler is deactivated when the agent or procedure enters a *Nextstate-node*, or enters a stopping condition or ceases to exist.

h) *Handle*

The exception handler for the current *Handle-node* is activated whenever interpretation the *Transition* of *Handle-node* is started in the agent, operation or procedure. The exception handler is deactivated when a *Nextstate-node* is entered in the agent, operation or procedure.

i) *Decision*

The exception handler is activated whenever interpretation of the given decision starts in the agent, operation or procedure. The exception handler is deactivated when the agent or procedure enters the transition of a decision branch (that is the exception handler covers the expression of the decision and whether the expression matches any of the ranges of the decision branches).

j) *Transition action (except decision)*

The exception handler is activated whenever interpretation of the given action is started in the agent, operation or procedure. The exception handler is deactivated when the agent or procedure interpretation of the action is complete.

k) *Transition terminator (with expressions)*

The exception handler is activated whenever the agent, operation or procedure enters the given terminator. The exception handler is deactivated when interpretation of the terminator is completed.

Any exception handler is deactivated when it handles an exception and creates an exception instance. The exception handlers for actions and terminators also cover the actions that result from the model for <transition>, for example <import expression>.

NOTE – The rules above imply that in some cases, several exception handlers may be deactivated at the same time. For example, if an exception handler for a state and one for an associated input transition are active at the same time, both exception handlers are deactivated when the input transition enters a nextstate node. Implicit states or stimuli are covered by exception handlers of the syntactical context; that is, <on exception>s or <on exception association area>s are copied into the model.

Model

When several <exception handler>s contain the same <exception handler name>, these <exception handler>s are concatenated into one <exception handler> having that <exception handler name>.

In a specialization, the association with the exception handler is considered as a part of the graph or the transition. If a virtual transition is redefined, the new transition replaces an <on exception> of the original transition. If a graph or a state is inherited in a specialization, any associated exception handler is inherited as well.

11.16.3 Handle

Abstract grammar

Handle-node	::	*Exception-identifier*
		[*Variable-identifier*]*
		[*On-exception*]
		Transition
Else-handle-node	::	[*On-exception*]
		Transition

The length of the list of optional *Variable-identifier*s in *Handle-node* must be the same as the number of *Sort-reference-identifier*s in the *Exception-definition* denoted by the *Exception-identifier*.

The sorts of the variables must correspond by position to the sorts of the data items that can be carried by the exception.

Concrete textual grammar

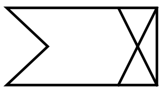
```
<handle> ::=
                    handle [<virtuality>] <exception stimulus list> <end>
                    [<on exception>] <transition>

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

<exception stimulus> ::=
                    <exception identifier> [ ( [ <variable> ] { , [ <variable> ] }* ) ]

<asterisk exception stimulus list> ::=
                    <asterisk>
```

When the <exception stimulus list> contains one <exception stimulus>, the <handle> represents a *Handle-node*. A <handle> with <asterisk exception stimulus list> represents an *Else-handle-node*.

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

Concrete graphical grammar

```
<handle area> ::=
                    <handle symbol> contains { [<virtuality>] <exception stimulus list> }
                    [ is connected to <on exception association area> ]
                    is followed by <transition area>

<handle symbol> ::=
```



The path to <transition area> in <handle area> must originate in <handle symbol>.

A <handle area>, whose <exception stimulus list> contains one <exception stimulus>, corresponds to one *Handle-node*. Each of the <exception identifier> contained in a <handle symbol> gives the name of one of the *Handle-node*s which this <handle symbol> represents. A <handle area> with <asterisk exception stimulus list> represents an *Else-handle-node*.

Semantics

A *Handle-node* consumes an instance of the specified exception type. The consumption of the exception instance makes the information conveyed by the exception instance available to the agent or procedure. The variables mentioned in the *Handle-node* are assigned the data items conveyed by the consumed exception instance.

The data items are assigned to the variables from left to right. If no variable is mentioned for a given parameter position in the exception, the data item at this position is discarded. If no data item is associated with a given parameter position, the corresponding variable becomes "undefined".

The **sender** expression is given the same result as the **self** expression.

NOTE – The **state** expression does not change to the name of the exception handler.

Model

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

When one or more of the <variable>s of a certain <exception stimulus> are <indexed variable>s or <field variable>s, all the <variable>s are replaced by unique, new, implicitly declared <variable identifier>s. Immediately before the <transition> of the <handle>, a <task> is inserted which in its <textual task body> contains an <assignment> for each of the <variable>s, assigning the result of the corresponding new variable to the <variable>. The results are assigned in the order from left to right of the list of <variable>s. This <task> becomes the first <action statement> in the <transition>.

A <handle> or <handle area> which contains <virtuality> is called a virtual handle transition. Virtual handle transitions are further described in 8.3.3.

12 Data

The concept of data in SDL is defined in this clause. This includes the data terminology, the concepts to define new data types and the predefined data.

Data in SDL is principally concerned with data types. A data type defines a set of elements or data items, referred to as sort, and a set of operations which can be applied to these data items. The sorts and operations define the properties of the data type. These properties are defined by data type definitions.

A data type consists of a set, which is the *sort* of the data type, and one or more *operations*. As an example, consider the predefined data type Boolean. The sort Boolean of the data type Boolean consists of the elements true and false. Among the operations of the data type Boolean are "=" (equal), "/=" (not equal), "not", "and", "or", "xor", and "=>" (implies). As a further example, consider the predefined data type Natural. It has the sort Natural consisting of the elements 0, 1, 2, etc., and the operations "=", "/=", "+", "-", "*", "/", "mod", "rem", "<", ">", "<=", ">=", and power.

SDL provides several predefined data types, which are familiar in both their behaviour and syntax. The predefined data types are described in Annex D.

Variables are objects that can be associated with an element of a sort by assignment. When the variable is accessed, the associated data item is returned.

The elements of the sort of a data type are either *values*, *objects* which are references to values, or *pids*, which are references to agents. The sort of a data type may be defined in the following ways:

a) Explicitly enumerating the elements of the sort.

b) Forming the Cartesian product of sorts S_1, S_2, … S_n; the sort is equal to the set that consists of all tuples that can be formed by taking the first element from sort S_1, taking the second element from sort S_2, … , and finally, taking the last element from sort S_n.

c) The sorts of pids are defined by defining an interface (see 12.1.2).

d) Several sorts are predefined and form the basis of the predefined data types described in Annex D. The predefined sorts Any and Pid are described in 12.1.5 and 12.1.6.

If the elements of a sort are objects, the sort is an object sort. If the elements of a sort are pids, the sort is a pid sort. The elements of a value sort are values.

Operations are defined from and to elements of sorts. For instance, the application of the operation for summation ("+") from and to elements of the Integer sort is valid, whereas summation of elements of the Boolean sort is not.

Each data item belongs to exactly one sort. That is, sorts never have data items in common.

For most sorts there are literal forms to denote elements 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 data item; for example, 12 and 012 can be used to denote the same Integer data item. 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 literal forms to denote the elements of the sort; for example, the sorts can also be formed as the Cartesian product of other sorts. In that case, the elements of these sorts are denoted by operations that construct the data item from elements of the component sort(s).

An expression denotes a data item. If an expression does not contain a variable or an imperative expression, e.g. if it is a literal of a given sort, each occurrence of the expression will always denote the same data item. These "passive" expressions correspond to a functional use of the language.

An expression that contains variables or imperative expressions may be interpreted as having different results during the interpretation of an SDL system depending on the data item associated with the variables. The active use of data includes assignment to variables, use of variables, and initialization of variables. The difference between active and passive expressions is that the result of a passive expression is independent of when it is interpreted, whereas an active expression may have different results depending on the current values, objects, or pids associated with variables or the current system state.

12.1 Data definitions

Data definitions are used to define data types. The basic mechanisms to define data are data type definitions (see 12.1.1) and interfaces (see 12.1.2). Specialization (see 12.1.3) allows the definition of a data type to be based on another data type, referred to as its supertype. The definition of the sort of the data type as well as operations implied for the sort are given by data type constructors (see 12.1.7). Additional operations can be defined as described in 12.1.4. Subclause 12.1.8 shows how to define the behaviour of the operations of a data type.

Since predefined data is defined in a predefined and implicitly used package Predefined (see 7.2 and D.3), the predefined sorts (for example, Boolean and Natural) and their operations may be freely used throughout the system. The semantics of Equality (12.2.5), Conditional expressions (12.2.6), and Syntypes (12.1.9.4) rely on the definition of the Boolean data type (see D.3.1). The semantics of Name class (see 12.1.9.1) also relies on the definition of Character and Charstring (see D.3.2 and D.3.4).

Abstract grammar

Data-type-definition	=	*Value-data-type-definition*
	|	*Object-data-type-definition*
	|	*Interface-definition*
Value-data-type-definition	::	*Sort*
		Data-type-identifier
		*Literal-signature**-set***
		*Static-operation-signature**-set***
		*Dynamic-operation-signature**-set***
Object-data-type-definition	::	*Sort*
		Data-type-identifier
		*Literal-signature**-set***
		*Static-operation-signature**-set***
		*Dynamic-operation-signature**-set***
Interface-definition	::	*Sort*
		*Data-type-identifier**

Data-type-identifier	=	*Identifier*
Sort-reference-identifier	=	*Sort-identifier*
	|	*Syntype-identifier*
	|	*Expanded-sort-identifier*
	|	*Reference-sort-identifier*
Sort-identifier	=	*Identifier*
Expanded-sort-identifier	=	*Sort-identifier*
Reference-sort-identifier	=	*Sort-identifier*
Sort	=	*Name*

A *Data-type-definition* introduces a sort that is visible in the enclosing scope unit in the abstract syntax. It may additionally introduce a set of literals and operations.

Concrete textual grammar

```
<data definition> ::=
                        <data type definition>
                  |     <interface definition>
                  |     <syntype definition>
                  |     <synonym definition>
```

A data definition represents a *Data-type-definition* if it is a <data type definition>, <interface definition>, or <syntype definition>.

```
<sort> ::=
                        <basic sort>
                  |     <anchored sort>
                  |     <expanded sort>
                  |     <reference sort>
                  |     <pid sort>

<basic sort> ::=
                        <sort identifier>
                  |     <syntype>

<anchored sort> ::=
                        this [<basic sort>]

<expanded sort> ::=
                        value <basic sort>

<reference sort> ::=
                        object <basic sort>

<pid sort> ::=
                        <sort identifier>
```

An <anchored sort> with <basic sort> is only allowed within the definition of <basic sort>.

An <anchored sort> is legal concrete syntax only if it occurs within a <data type definition>. The <basic sort> in the <anchored sort> must name the <sort> introduced by the <data type definition>.

Semantics

A data definition is used either for the definition of a data type or interface or the definition of a synonym for an expression as further defined in 12.1, 12.1.9.4, or 12.1.9.6.

Each <data type definition> introduces a sort with the same name as the <data type name> (see 12.1.1). Each <interface definition> introduces a sort with the same name as the <interface name> (see 12.1.2).

NOTE 1 – To avoid cumbersome text, the convention is used that the phrase "the sort S" is often used instead of "the sort defined by the data type S" or "the sort defined by the interface S" when no confusion is likely to arise.

A <sort identifier> names a <sort> introduced by a data type definition.

A sort is a set of elements: values, objects (that is, references to values) or pids (that is, references to agents). Two different sorts have no elements in common. A <value data type definition> introduces a sort that is a set of values. An <object data type definition> introduces a sort that is a set of objects. An <interface definition> introduces a pid sort.

If a <sort> is an <expanded sort>, then variables, synonyms, fields, parameters, return, signals, timers, and exceptions defined with that <sort> will be associated with values of the sort rather than with references to these values, even if the sort has been defined as a set of objects. An *Expanded-sort-identifier* is represented by an <expanded sort>.

If a <sort> is a <reference sort>, then variables, synonyms, fields, parameters, return, signals, timers, and exceptions defined with that <sort> will be associated with references to values of the sort rather than with values of the sort, even if the sort has been defined as a set of values. A *Referenced-sort-identifier* is represented by a <reference sort>.

The meaning of an <anchored sort> is given in 12.1.3.

The <sort identifier> in a <pid sort> must reference a pid sort.

Model

An <expanded sort> with a <basic sort> that represents a value sort is replaced by the <basic sort>.

A <reference sort> with a <basic sort> that represents an object sort is replaced by the <basic sort>.

NOTE 2 – As a consequence, the keyword **value** has no effect if the sort has been defined as a set of values, and the keyword **object** has no effect if the sort has been defined as a set of objects.

An <anchored sort> without a <basic sort> is a shorthand for specifying a <basic sort> referencing the name of the data type definition or syntype definition in the context of which the <anchored sort> occurs.

12.1.1 Data type definition

Concrete textual grammar

```
<data type definition> ::=
                        {<package use clause>}*
                        <type preamble> <data type heading> [<data type specialization>]
                        {       <end> [ <data type definition body> <data type closing> <end>]
                            | <left curly bracket> <data type definition body> <right curly bracket> }

<data type definition body> ::=
                        {<entity in data type>}* [<data type constructor>] <operations>
                        [<default initialization>[ <end> ] ]

<data type closing> ::=
                        { endvalue | endobject } type [<data type name>]

<data type heading> ::=
                        { value | object } type <data type name>
                            [ <formal context parameters> ] [<virtuality constraint>]

<entity in data type> ::=
                        <data type definition>
                   |    <syntype definition>
                   |    <synonym definition>
                   |    <exception definition>

<operations> ::=
                        <operation signatures>
                        <operation definitions>

<data type reference> ::=
                        <type preamble>
                        { value | object } type <data type identifier> <type reference properties>
```

A <value data type definition> contains the keyword value in <data type heading>. An <object data type definition> contains the keyword object in <data type heading>.

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

The keyword **value** or **object** and <data type name> in a <data type closing> must be matched by the keywords **endvalue** or **endobject**, respectively, and name in the corresponding <data type heading>, if present.

Concrete graphical grammar

```
<data type reference area> ::=
                    <type reference area>
```

The <type reference area> that is part of a <data type reference area> must have a <graphical type reference heading> that contains a <data type name>.

Semantics

A <data type definition> consists of a <data type constructor> which describes the elements of the sort (see 12.1.6) and operations induced by the way the sort is constructed, and <operations> which defines a set of operations that can be applied to the elements of a sort (see 12.1.4). A data type may also be based on a supertype through specialization (see 12.1.3).

12.1.2 Interface definition

Interfaces are defined in packages, agents or agent types. An interface defines a pid sort, which has elements that are references to agents.

An interface may define signals, remote procedures, remote variables, and exceptions. The defining context of entities defined in the interface is the scope unit of the interface, and the entities defined are visible where the interface is visible. An interface may also refer to signals, remote procedures, or remote variables defined outside the interface by the <interface use list>.

An interface is used in a signal list to denote that the signals, remote procedure calls, and remote variables of the interface definition are included in the signal list.

Concrete textual grammar

```
<interface definition> ::=
                    {<package use clause>}*
                    [<virtuality>] <interface heading>
                    [<interface specialization>]
                        <end> <entity in interface>* [<interface use list>]
                    <interface closing>
              |     {<package use clause>}*
              |     [<virtuality>] <interface heading>
                    [<interface specialization>] <end>
              |     {<package use clause>}*
              |     [<virtuality>] <interface heading>
                    [<interface specialization>] <left curly bracket>
                        <entity in interface>* [<interface use list>]
                    <right curly bracket>

<interface closing> ::=
                    endinterface [<interface name>] <end>

<interface heading> ::=
                    interface <interface name>
                        [<formal context parameters>] [<virtuality constraint>]
```

<entity in interface> ::=
<signal definition>
| <interface variable definition>
| <interface procedure definition>
| <exception definition>

<interface use list> ::=
use <signal list> <end>

<interface variable definition> ::=
dcl <remote variable name> { **,** <remote variable name>}* <sort> <end>

<interface procedure definition> ::=
procedure <remote procedure name> <procedure signature> <end>

<interface reference> ::=
[<virtuality>]
interface <interface identifier> <type reference properties>

The <signal list> in an <interface definition> shall only contain <signal identifier>s, <remote procedure identifier>s, <remote variable identifier>s and <signal list identifier>s. If a <signal list identifier> is part of the <signal list> it must also respect this restriction.

The <formal context parameters> shall only contain <signal context parameter>, <remote procedure context parameter>, <remote variable context parameter>, <sort context parameter> or <exception context parameter>.

The defining context of entities defined in the interface (<entity in interface>) is the scope unit of the interface, and the entities defined are visible where the interface is visible.

Concrete graphical grammar

<interface reference area> ::=
<type reference area>

The <type reference area> that is part of an <interface reference area> must have a <graphical type reference heading> that contains an <interface name>**.**

Semantics

The semantics of <virtuality> is defined in 8.3.2.

The inclusion of an <interface identifier> in a <signal list> means that all signal identifiers, remote procedure identifiers and remote variable identifiers forming part of the <interface definition> are included in the <signal list>.

Model

Interfaces are implicitly defined by the agent, its state machine and agent type definitions. The implicitly defined interface has the same name as the agent or agent type that defined it.

NOTE 1 – Because every agent and agent type has an implicitly defined interface with the same name, any explicitly defined interface must have a different name from every agent and agent type defined in the same scope, otherwise there are name clashes.

The interface defined by a state machine contains in its <interface specialization> all interfaces given in the incoming signal list associated with explicit or implicit gates of the state machine. The interface also contains in its <interface use list> all signals, remote variables and remote procedures given in the incoming signal list associated with explicit or implicit gates of the state machine.

The interface defined by an agent or agent type contains in its <interface specialization> the interface defined by the composite state representing its state machine.

The interface defined by a type based agent contains in its <interface specialization> the interface defined by its type.

NOTE 2 – To avoid cumbersome text, the convention is used that the phrase "the pid sort of the agent A" is often used instead of "the pid sort defined by the interface implicitly defined by the agent A" when no confusion is likely to arise.

12.1.3 Specialization of data types

Specialization allows the definition of a data type based on another (super) type. The sort defined by the specialization is considered a subsort of the sort defined by the base type. The sort defined by the base type is a supersort of the sort defined by the specialization.

Concrete textual grammar

```
<data type specialization> ::=
                    inherits <data type type expression> [<renaming>] [adding]

<interface specialization> ::=
                    inherits <interface type expression> { , <interface type expression> }* [adding]

<renaming> ::=
                    ( <rename list> )

<rename list> ::=
                    <rename pair> { , <rename pair> }*

<rename pair> ::=
                    <operation name> <equals sign> <base type operation name>
               |    <literal name> <equals sign> <base type literal name>
```

The *Data-type-identifier* in the *Data-type-definition* represented by the <data type definition> in which <data type specialization> (or <interface specialization>) is contained identifies the data type represented by the <data type type expression> in its <data type specialization> (see also 8.1.2).

An *Interface-definition* may contain a list of *Data-type-identifier*s. The interface denoted by an *Interface-definition* is a specialization of all the interfaces denoted by the *Data-type-identifier*s.

The resulting content of a specialized interface definition (<interface specialization>) consists of the content of the supertypes followed by the content of the specialized definition. This implies that the set of signals, remote procedures and remote variables of the specialized interface definition is the union of those given in the specialized definition itself and those of the supertypes. The resulting set of definitions must obey the rules given in 6.3.

The <data type constructor> must be of the same kind as the <data type constructor> used in the <data type definition> of the sort referenced by <data type type expression> in the <data type specialization>. That is, if the <data type constructor> used in a (direct or indirect) supertype was a <literal list> (<structure definition>, <choice definition>), then the <data type constructor> must also be a <literal list> (<structure definition>, <choice definition>).

Renaming can be used to change the name of inherited literals, operators, and methods in the derived data type.

All <literal name>s and all <base type literal name>s in a <rename list> must be distinct.

All <operation name>s and all <base type operation name>s in a <rename list> must be distinct.

A <base type operation name> specified in a <rename list> must be an operation with <operation name> defined in the data type definition defining the <base type> of <data type type expression>.

Semantics

Sort compatibility determines when a sort can be used in place of another sort, and when it cannot. This relation is used for assignments (see 12.3.3), for parameter passing (see 12.2.7 and 9.4), for re-declaration of parameter types during inheritance (see 12.1.2), and for actual context parameters (see 8.1.2).

Let T and V be two sorts. V is sort compatible to T if and only if either:

a) V and T are the same sort;

b) V is directly sort compatible to T;

c) T has been defined by an object type or an interface and, for some sort U, V is sort compatible to U and U is sort compatible to T.

NOTE 1 – Sort compatibility is transitive only for sorts defined by object types or interfaces, but not for sorts defined by value types.

Let T and V be sorts. V is directly sort compatible to T if and only if either:

a) V is denoted by a <basic sort> and T is an object sort and T is a supersort of V;

b) V is denoted by an <anchored sort> of the form **this** T;

c) V is denoted by a <reference sort> of the form **object** T;

d) T is denoted by a <reference sort> of the form **object** V;

e) V is denoted by an <expanded sort> of the form **value** T;

f) T is denoted by a <expanded sort> of the form **value** V; or

g) V is denoted by a <pid sort> (see 12.1.2) and T is a supersort of V.

Model

The model for specialization in 8.2.14 is used, augmented as follows.

A specialized data type is based on another (base) data type by using a <data type definition> in combination with a <data type specialization>. The sort defined by the specialization is disjoint from the sort defined by the base type.

If the sort defined by the base type has literals defined, the literal names are inherited as names for literals of the sort defined by the specialized type 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 <rename pair> in which case the literal is renamed to the first name in that pair.

If the base type has operators or methods defined, the operation names are inherited as names for operators or methods of the sort being defined, subject to the restrictions stated in 8.3.1, unless the operator or method has been declared as private (see 12.1.9.3) or operation renaming has taken place for that operator or method. Operation renaming has taken place for an operator or method if the inherited operation name appears as the second name in a <rename pair> in which case the operator or method is renamed to the first name in that pair.

When several operators or methods of the <base type> of <sort type expression> have the same name as the <base type operation name> in a <rename pair>, then all of these operators or methods are renamed.

In every occurrence of an <anchored sort> in the specialized type, the <basic sort> is replaced by the subsort.

The argument sorts and result of an inherited operator or method are the same as those of the corresponding operator or method of the base type, except that in every <argument> containing an <anchored sort> in the inherited operator or method the <basic sort> is replaced the subsort. For inherited virtual methods, <argument virtuality>is added to an <argument> containing an <anchored sort>, if it is not already present.

NOTE 2 – According to the model for specialization in 8.2.14, an operator is only inherited if its signature contained at least one <anchored sort> or renaming had taken place.

12.1.4 Operations

Abstract grammar

Dynamic-operation-signature	=	*Operation-signature*
Static-operation-signature	=	*Operation-signature*
Operation-signature	::	*Operation-name*
		*Formal-argument**
		[*Result*]
Operation-name	=	*Name*
Formal-argument	=	*Virtual-argument*
	|	*Nonvirtual-argument*
Virtual-argument	::	*Argument*
Nonvirtual-argument	::	*Argument*
Argument	=	*Sort-reference-identifier*

The notion of sort compatibility is extended to *Operation-signature*s. An *Operation-signature* S1 is sort compatible to an *Operation-signature* S2 when:

a) S1 and S2 have the same number of Formal-arguments; and

b) for every *Virtual-argument* A of S1, the sort identified by its *Sort-reference-identifier* is sort compatible to the sort identified by the *Sort-reference-identifier* of the corresponding argument in S2;

c) for every *Nonvirtual-argument* A of S1, the sort identified by its *Sort-reference-identifier* is the same sort as the sort identified by the *Sort-reference-identifier* of the corresponding argument in S2.

Concrete textual grammar

```
<operation signatures> ::=
                         [<operator list>] [<method list>]

<operator list> ::=
                         operators <operation signature> { <end> <operation signature> }* <end>

<method list> ::=
                         methods <operation signature> { <end> <operation signature> }* <end>

<operation signature> ::=
                         <operation preamble>
                         { <operation name> | <name class operation> }
                              [<arguments>] [<result>] [<raises>]

<operation preamble> ::=
                         [<virtuality>] [<visibility>] | [<visibility>] [<virtuality>]

<operation name> ::=
                         <operation name>
                    |    <quoted operation name>

<arguments> ::=
                         ( <argument> { , <argument> }* )

<argument> ::=
                         [<argument virtuality>] <formal parameter>

<formal parameter> ::=
                         <parameter kind> <sort>

<argument virtuality> ::=
                         virtual

<result> ::=
                         <result sign> <sort>
```

In an *Operation-signature*, each *Sort-reference-identifier* in *Formal-argument* is represented by an argument <sort>, and the *Result* is represented by the result <sort>. A <sort> in an <argument> that contains <argument virtuality> represents a *Virtual-argument*, otherwise the <sort> of the <argument> represents a *Nonvirtual-argument*.

The *Operation-name* is unique within the defining scope unit in the abstract syntax even though the corresponding <operation name> may not be unique. The unique *Operation-name* is derived from:

a) the <operation name>; plus

b) the (possible empty) list of argument sort identifiers; plus

c) the result sort identifier; plus

d) the sort identifier of the data type definition in which the <operation name> is defined.

<quoted operation name> allows for operator and method names that have special syntactic forms. The special syntax is introduced so that, for example, arithmetic operations and Boolean operations can have their usual syntactic form. That is, the user can write "(1 + 1) = 2" rather than having to use, for example, equal(add(1,1),2).

If <operation signature> is contained in an <operator list>, then the <operation signature> represents a *Static-operation-signature*, and the <operation signature> must not contain <virtuality> or <argument virtuality>.

If <operation signature> is contained in a <method list> and <virtuality> is not present, then the <operation signature> represents a *Static-operation-signature* and none of the <argument>s must contain <argument virtuality>.

If <operation signature> is contained in a <method list> and <virtuality> is present, then the <operation signature> represents a *Dynamic-operation-signature*. In this case, a set of *Dynamic-operation-signature*s is formed consisting of the *Dynamic-operation-signature* represented by the <operation signature> and any element in the signature set of the matching method in the supertype with an *Operation-name* derived from the same <operation name> taking renaming into account, and such that the *Operation-signature* is sort compatible to the *Operation-signature* in the supertype, if any.

This set must be closed in the following sense: for any two *Operation-signature*s S_i and S_j in the set of *Operation-signature*s, the unique *Operation-signature* S such that:

a) S is sort compatible to S_i and S_j; and

b) for any *Operation-signature* S_k that is sort compatible to both S_i and S_j, S_k is also sort compatible to S,

is also in the set of *Dynamic-operation-signature*s.

This condition ensures that the set of *Dynamic-operation-signature*s forms a lattice and guarantees that a unique best matching *Operation-signature* can be found when interpreting an operation application (see 12.2.7). If the set of *Dynamic-operation-signature*s does not satisfy this condition, the <sdl specification> is invalid.

NOTE – Specialization of a type may require that additional *Operation-signature*s be added to the <method list> to satisfy this condition.

<result> in <operation signature> may be omitted only if the <operation signature> occurred in a <method list>.

<argument virtuality> is legal only if <virtuality> contained the keywords **virtual** or **redefined**.

Semantics

The quoted forms of infix or monadic operators or methods are valid names for operators or methods.

An operator or method has a result sort, which is the sort identified by the result.

Model

If <operation signature> is contained in a <method list> this is derived syntax and is transformed as follows: an <argument> is constructed from the keyword **virtual**, if <virtuality> was present, the <parameter kind> **in/out**, and the <sort identifier> of the sort being defined by the enclosing <data type definition>. If there are no <arguments>, then <arguments> is formed from the constructed <argument> and inserted into the <operation signature>. If there are <arguments>, the constructed <argument> is added to the start of the original list of <argument>s in the <arguments>.

If the <sort> of an <argument> is an <anchored sort>, the <argument> implicitly contains <argument virtuality>. If an <operation signature> contains the keywords **redefined** in <virtuality>, for every <argument> in the matching <operation signature> of the base type, if this <argument> (implicitly or explicitly) contains <argument virtuality>, then the corresponding <argument> in <operation signature> implicitly also contains <argument virtuality>.

An <argument> without an explicit <parameter kind> has the implicit <parameter kind> **in**.

12.1.5 Any

Every value or object type is (directly or indirectly) a subtype of the abstract object type Any. When a variable is declared to be of sort Any, data items belonging to any value or object sort may be assigned to that variable.

Any is implicitly defined by the following <data type definition>, where Boolean is the predefined Boolean sort:

```
abstract object type Any
operators
    equal             ( this Any, this Any    ) -> Boolean;
    clone             ( this Any              ) -> this Any;
methods
    virtual is_equal  ( this Any              ) -> Boolean;
    virtual copy      ( this Any              ) -> this Any;
endobject type Any;
```

NOTE 1 – Because all data type constructors implicitly redefine the virtual methods of the data type Any, these methods cannot be explicitly redefined in a <data type definition>.

In addition, each <object data type definition> introducing a sort named S implies *Operation-signature*s equivalent to including the explicit definition in the following <operation signature>s in the <operator list>:

```
Null          -> this S;
Make          -> this S;
```

as well as the following <operation signature> in the <method list>:

```
virtual finalized -> <<S>> Null;
```

Concrete textual grammar

The data type Any can be qualified by **package** Predefined.

Semantics

The operators and methods defined by Any are available to any value or object data type.

Each <object data type definition> adds a unique data item denoting a reference that has not yet been associated with a value. The operator Null returns this data item. Any attempt to obtain an associated value from the object returned by Null will raise the predefined exception InvalidReference (see D.3.16).

The operator Make introduced by an <object data type definition> creates a new, uninitialized element of the <result sort> of the Make operator. Each <object data type definition> provides an appropriate definition for the Make operator.

The operator *equal* compares two values for equality (when defined by a value type), or compares two values referenced by objects for equality (when defined by an object type). Let X and Y be the results of its actual parameter *Expression*s, then:

a) if either X or Y is Null, the result is the predefined Boolean value true if both are Null, and the predefined Boolean value false if only one is Null; otherwise

b) if the dynamic sort of Y is not sort compatible to the dynamic sort of X, the result is the predefined Boolean value false; otherwise

c) the result is obtained by interpreting *x.is_equal*(*y*), where *x* and *y* represent X and Y, respectively.

The operator *clone* creates a new data item belonging to the sort of its actual parameter and initializes the newly created data item by applying copy to that data item, given the original actual parameter. After applying *clone,* the newly created data item is equal to the actual parameter. Let Y be the result of its actual parameter *Expression*, then the operator *clone* is defined as:

a) if Y is Null, the result is Null; otherwise

b) if the sort of X is an object sort, let X be the result of interpreting the *Make* operator for the data type that defined the sort of Y. The result is obtained by interpreting *x.copy*(*y*), where *x* and *y* represent X and Y, respectively; otherwise

c) if the sort of X is a value sort, let X be an arbitrary element of the sort of X. The result is obtained by interpreting *x.copy*(*y*), where *x* and *y* represent X and Y, respectively.

The method *is_equal* compares **this** to the actual parameter, component by component, if there are any. In general, for the *is_equal* method to return the predefined Boolean value true, neither **this** nor the actual parameter must be Null, and the sort of the actual parameter must be sort compatible to the sort of **this**.

Data type definitions may redefine *is_equal* to account for differences in the semantics of their corresponding sorts. The type constructors will implicitly redefine *is_equal* as follows. Let X and Y be the results of its actual parameter *Expression*s, then:

a) if the sort of X has been constructed by a <literal list>, then the result is the predefined Boolean value true if X and Y are the same value;

b) if the sort of X has been constructed by a <structure definition>, then the result is the predefined Boolean value true if for every component of X, this component is equal to the corresponding component of Y as determined by interpreting an *Equality-expression* with these components as the operands, omitting those components of X and Y that have been defined as optional and which are not present.

The method *copy* copies the actual parameter onto **this**, component by component, if there are any. Each data type constructor adds a method that redefines *copy*. In general, neither **this** nor the actual parameter must be Null, and the sort of the actual parameter must be sort compatible to the sort of **this**. Every redefinition of *copy* must satisfy the post-condition that after application of the *copy* method, **this** *is_equal* to the actual parameter.

Data type definitions may redefine *copy* to account for differences in the semantics of their corresponding sorts. The type constructors will automatically redefine *copy* as follows. Let X and Y be the results of its actual parameter *Expression*s, then:

a) if the sort of X has been constructed by a <literal list>, Y is copied onto X;

b) if the sort of X has been constructed by a <structure definition>, for every component of X, the corresponding component of Y is copied onto that component of X by interpreting *xc.modify(yc)* – where *xc* represents the component of X, *modify* is the field modify method for this component, and *yc* represents the corresponding component of Y, omitting those components of X and Y that have been defined as optional and which are not present.

NOTE 2 – The interpretation of the *modify* method involves an assignment of the actual parameter to the formal parameter and, consequently, a recursive call to the copy method (see 12.3.3).

Model

If a <data type definition> does not contain an <data type specialization>, this is a shorthand notation for a <data type definition> with a <data type specialization>

```
        inherits Any;
```

12.1.6 Pid and pid sorts

Every interface is (directly or indirectly) a subtype of the interface Pid. When a variable is declared to be of sort Pid, data items belonging to any pid sort may be assigned to that variable.

Concrete textual grammar

The data type Pid can be qualified by **package** Predefined.

Semantics

The sort Pid contains a single data item denoted by the literal Null. Null represents a reference that is not associated with any agent.

An *Interface-definition* represented by an <interface definition> without an <interface specialization> contains only a *Data-type-identifier* denoting the interface Pid.

An element of a pid sort introduced by an interface implicitly defined by an agent definition is associated with a reference to the agent by the interpretation of a *Create-request-node* (see 11.13.2).

Each interface adds a compatibility check operation that, given a signal, will determine whether either:

a) the signal is defined or used in the interface; or

b) the signal is contained on an inwards gate connected to the state machine of the agent that implicitly defined the interface; or

c) the compatibility check is satisfied for a pid sort defined by an interface contained in its <interface specialization>.

If this is not fulfilled, then the predefined exception InvalidReference (see D.3.16) shall be raised. The compatibility check is defined similarly for remote variables (see 10.6) and remote procedures (see 10.5).

NOTE – A pid sort can be polymorphically assigned (see 12.3.3).

12.1.7 Data type constructors

Data type constructors specify the contents of the sort of a data type, either by enumerating the elements that constitute the sort or by collecting all data items which can be obtained by constructing a tuple from elements of given sorts.

Concrete textual grammar

```
<data type constructor> ::=
                        <literal list>
                |       <structure definition>
                |       <choice definition>
```

12.1.7.1 Literals

The literal data type constructor specifies the contents of the sort of a data type by enumerating the (possibly infinitely many) elements of the sort. The literal data type constructor implicitly defines operations that allow comparison between the elements of the sort. The elements of a literal sort are called literals.

Abstract grammar

Literal-signature	::	*Literal-name*
		Result
Literal-name	=	*Name*

Concrete textual grammar

```
<literal list> ::=
                    [<visibility>] literals <literal signature> { , <literal signature> }* <end>

<literal signature> ::=
                    <literal name>
               |    <name class literal>
               |    <named number>

<literal name> ::=
                    <literal name>
               |    <string name>

<named number> ::=
                    <literal name> <equals sign> <Natural simple expression>
```

In a *Literal-signature*, the *Result* is the sort introduced by the <data type definition> defining the <literal signature>.

The *Literal-name* is unique within the defining scope unit in the abstract syntax even though the corresponding <literal name> may not be unique. The unique *Literal-name* is derived from:

a) the <literal name>; plus

b) the sort identifier of the data type definition in which the <literal name> is defined.

NOTE – A <string name> is one of the lexical units <character string>, <bit string>, and <hex string>.

Each result of <Natural simple expression> occurring in a <named number> must be unique among all <literal signature>s in the <literal list>.

Semantics

A <literal list> defines a sort by enumerating all the elements of the set. Each element in the sort is represented by a *Literal-signature*.

Literals formed from <character string> are used for the predefined data sorts Charstring (see D.3.4) and Character (see D.3.2). They also have a special relationship with <regular expression>s (see 12.1.9.1). Literals formed from <bit string> and <hex string> are also used for the predefined data sort Integer (see D.3.5). These literals may also be defined to have other uses.

A <literal list> redefines the operations (directly or indirectly) inherited from Any as described in 12.1.5.

The meaning of <visibility> in <literal list> is explained in 12.1.9.3.

Model

A <literal name> in a <literal list> is derived syntax for a <named number> containing the <literal name> and containing a <Natural simple expression> denoting the lowest possible non-negative Natural value not occurring in any other <literal signature>s of the <literal list>. The replacement of <literal name>s by the <named number>s takes place one by one from left to right.

A literal list is derived syntax for the definition of operators that establish an ordering of the elements in the sort defined by the <literal list>:

a) operators that compare two data items with respect to the established ordering;

b) operators that return the first, last, next, or previous data item in the ordering; and

c) an operator that gives the position of each data item in the ordering.

A <data type definition> introducing a sort named S by a <literal list> implies a set of *Static-operation-signature*s equivalent to the explicit definitions in the following <operator list>:

```
"<"   ( this S, this S )   -> Boolean;
">"   ( this S, this S )   -> Boolean;
"<="  ( this S, this S )   -> Boolean;
">="  ( this S, this S )   -> Boolean;
first                      -> this S;
last                       -> this S;
succ  ( this S )           -> this S;
pred  ( this S )           -> this S;
num   ( this S )           -> Natural;
```

where Boolean is the predefined Boolean sort and Natural is the predefined Natural sort.

The <literal signature>s in a <data type definition> are nominated in ascending order of the <Natural simple expression>s. For example,

```
    literals C = 3, A, B;
```

implies A<B and B<C.

The comparison operators "<" (">","<=",">=") represent the standard less-than (greater-than, less-or-equal-than, and greater-or-equal-than) comparison between the <Natural simple expression>s of two literals. The operator first returns the first data item in the ordering (the literal with the lowest <Natural simple expression>). The operator last returns the last data item in the ordering (the literal with the highest <Natural simple expression>). The operator pred returns the preceding data item, if one exists, or the last data item, otherwise. The operator succ returns the successor data item in the ordering, if one exists, or the first data item, otherwise. The operator num returns the Natural value corresponding to the <Natural simple expression> of the literal.

If <literal signature> is a <regular expression>, this is shorthand for enumerating a (possibly infinite) set of <literal name>s as described in 12.1.9.1.

12.1.7.2 Structure data types

The structure data type constructor specifies the contents of a sort by forming the Cartesian product of a set of given sorts. The elements of a structure sort are called structures. The structure data type constructor implicitly defines operations that construct structures from the elements of the component sorts, projection operations to access the component elements of a structure, as well as operations to update the component elements of a structure.

Concrete textual grammar

```
<structure definition> ::=
                        [<visibility>] struct [<field list>] <end>

<field list> ::=
                        <field> { <end> <field> }*

<field> ::=
                        <fields of sort>
                  |     <fields of sort> optional
                  |     <fields of sort> <field default initialization>
```

<fields of sort> ::=
[<visibility>] <field name> { , <field name> }* <field sort>

<field default initialization> ::=
default <constant expression>

<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 elements are all the tuples that can be constructed from data items belonging to the sorts given in <field list>. An element of a structure sort has as many component elements as there are <field>s in the <field list>, although a field may not be associated with a data item, if the corresponding <field> had been declared with the keyword **optional**, or has not yet been initialized.

A <structure definition> redefines the operations (directly or indirectly) inherited from Any as described in 12.1.5.

The meaning of <visibility> in <fields of sort> and <structure definition> is explained in 12.1.9.3.

Model

A <field list> containing a <field> with a list of <field name>s in a <fields of sort> is derived concrete syntax where this <field> is replaced by a list of <field>s separated by <end>, such that each <field> in this list resulted from copying the original <field> and substituting one <field name> for the list of <field name>s, in turn for each <field name> in the list.

A structure definition is derived syntax for the definition of:

a) an operator, Make, to create structures;

b) methods to modify structures and to access component data items of structures; and

c) methods to test for the presence of optional component data items in structures.

The <arguments> for the Make operator contains the list of <field sort>s occurring in the field list in the order in which they occur. The result <sort> for the Make operator is the sort identifier of the structure sort. The Make operator creates a new structure and associates each field with the result of the corresponding formal parameter. If the actual parameter was omitted in the application of the Make operator, the corresponding field gets no value; that is, it becomes "undefined".

A <structure definition> introducing a sort named S implies a set of *Dynamic-operation-signatures* equivalent to the explicit definitions in the following <method list>, for each <field> in its <field list>:

virtual *field-modify-operation-name* (<field sort>) -> S;
virtual *field-extract-operation-name* -> <field sort>;
field-presence-operation-name -> Boolean;

where Boolean is the predefined Boolean sort, and <field sort> is the sort of the field.

The name of the implied method to modify a field, *field-modify-operation-name*, is the field name concatenated with "Modify". The implied method to modify a field associates the field with the result of its argument *Expression*. When <field sort> was an <anchored sort>, this association takes place only if the dynamic sort of the argument *Expression* is sort compatible with the <field sort> of this field. Otherwise, the predefined exception UndefinedField (see D.3.16) is raised.

The name of the implied method to access a field, *field-extract-operation-name*, is the field name concatenated with "Extract". The method to access a field returns the data item associated with that field. If, during interpretation, a field of a structure is "undefined", then applying the method to access this field to the structure leads to the raise of the predefined exception UndefinedField.

The name of the implied method to test for the presence of a field data item, *field-presence-operation-name*, is the field name concatenated with "Present". The method to test for the presence of a field data item returns the predefined Boolean value false if this field is "undefined", and the predefined Boolean value true otherwise. A method to test for the presence of a field data item is only defined if this <field> contained the keyword **optional**.

If a <field> is defined with a <field default initialization>, this is derived syntax for the definition of this <field> as optional. When a structure of this sort is created and no actual argument is provided for the default field, an immediate modification of the field by the associated <constant expression> after structure creation is added.

12.1.7.3 Choice data types

A choice data type constructor is a shorthand notation for defining a structure type with all components optional, and ensuring that every structure data item will always have exactly one component data item present. The choice data type thus simulates a sort that is a disjoint sum of the elements of the component sorts.

Concrete textual grammar

<choice definition> ::=
[<visibility>] **choice** [<choice list>] <end>

<choice list> ::=
<choice of sort> { <end> <choice of sort> }*

<choice of sort> ::=
[<visibility>] <field name> <field sort>

Each <field name> of a choice sort must be different from every other <field name> of the same <choice definition>.

Semantics

A <choice definition> redefines the operations (directly or indirectly) inherited from Any as described in 12.1.5.

The meaning of <visibility> in <choice of sort> and <choice definition> is explained in 12.1.9.3.

Model

A data type definition containing a <choice definition> is derived syntax and transformed in the following steps: let *Choice-name* be the <data type name> of the original data type definition, then:

a) A <value data type definition> with an anonymous name, *anon*, and a <structure definition> as the type constructor is added. In the <value data type definition>, for each <choice of sort>, a <field> is constructed containing the equivalent <fields of sort> with the keyword **optional**.

b) A <value data type definition> with an anonymous name, *anonPresent*, is added with a <literal list> containing all the <field name>s in the <choice list> as <literal name>s. The order of the literals is the same as the order in which the <field name>s were specified.

c) A <data type definition> with an anonymous name, *anonChoice*, is constructed as follows:

```
object type anonChoice
struct
        protected Present anonPresent;
        protected Choice anon;
endobject type anonChoice;
```

if the original data type definition had defined an object sort. Otherwise, the <data type definition> is a <value data type definition>.

d) A <data type definition> is constructed as follows:

```
object type Choice-name inherits anonChoice (anonMake = Make,
                anonPresentModify = PresentModify,
                anonPresentExtract = PresentExtract,
                anonChoiceModify = ChoiceModify,
                anonChoiceExtract = ChoiceExtract )
adding
        operations
endobject type Choice-name;
```

if the original data type definition had defined an object type, and where *operations* is <operations>, as defined below. Otherwise, the <data type definition> is a <value data type definition>. The <renaming> renames the mentioned operations inherited from *anonChoice* to anonymous names.

e) For each <choice of sort>, an <operation signature> is added to the <operator list> of *operations* representing an implied operator for creating data items:

```
field-name ( field-sort ) -> Choice-name;
```

where *field-name* is the <field name> and *field-sort* is the <field sort> in <choice of sort>. The implied operator for creating data items creates a new structure by calling *anonMake*, initializing the field Choice with a newly created structure initialized with <field name>, and assigning the literal corresponding to the <field name> to the field Present.

f) For each <choice of sort>, an <operation signature>s are added to the <method list> of *operations* representing implied methods for modifying and accessing data items:

```
virtual field-modify ( field-sort ) -> Choice-name;
virtual field-extract -> field-sort;
field-present -> Boolean;
```

where *field-extract* is the name of the method implied by *anon* to access the corresponding field, *field-modify* is the name of the method implied by *anon* to modify that field, and *field-present* is the name of the method implied by *anon* to test for the presence of a field data item. Calls to *field-extract* and *field-present* are forwarded to Choice. Calls to *field-modify* assign a newly created structure initialized with <field name> to Choice and assign the literal corresponding to the <field name> to Present.

g) An <operation signature> is added to the <operator list> of *operations* representing an implied operator for obtaining the sort of the data item currently present in Choice:

```
PresentExtract ( Choice-name ) -> anonPresent;
```

PresentExtract returns the value associated with the Present field.

12.1.8 Behaviour of operations

A <data type definition> allows operations to be added to a data type. The behaviour of operations can be defined in a manner similar to value returning procedure calls. However, the operations of a data type must not access or change the global state of the input queues of the agents in which they are called. They therefore only contain a single transition.

Concrete textual grammar

```
<operation definitions> ::=
                              {     <operation definition>
                              |     <textual operation reference>
                              |     <external operation definition> }*

<operation definition> ::=
                              {<package use clause>}*
                              <operation heading> <end>
                                    <entity in operation>*
                                    <operation body>
                              { endoperator | endmethod } [ [<qualifier>] <operation name>] <end>
                        |     {<package use clause>}*
                              <operation heading>
                                    [ <end> <entity in operation>+ ] <left curly bracket>
                                          <statement list>
                                     <right curly bracket>

<operation heading> ::=
                              { operator | method } <operation preamble> [<qualifier>] <operation name>
                                    [<formal operation parameters>]
                                    [<operation result>] [<raises>]

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

<formal operation parameters> ::=
                              ( <operation parameters> {, <operation parameters> }* )

<operation parameters> ::=
                              [<argument virtuality>] <parameter kind> <variable name> {, <variable name>}* <sort>

<entity in operation> ::=
                              <data definition>
                        |     <variable definition>
                        |     <exception definition>
                        |     <select definition>
                        |     <macro definition>

<operation body> ::=
                              [<on exception>] <start> { <free action> | <exception handler> }*

<operation result> ::=
                              <result sign> [<variable name>] <sort>

<textual operation reference> ::=
                              <operation heading> referenced <end>

<external operation definition> ::=
                              <operation heading> external <end>
```

The keywords **operator** or **method** used in <operation heading> must be matched in the corresponding <operation definition> by **endoperator** and **endmethod**, respectively.

There is no corresponding graphical syntax for <external operation definition>.

<formal operation parameters> and <operation result> in <textual operation reference> and <external operation definition> may be omitted if there is no other <textual operation reference> or <external operation definition>, respectively, within the same sort which has the same name. In this

case, the <formal operation parameters> and the <operation result> are derived from the <operation signature>.

For each <operation signature> at most one corresponding <operation definition> can be given.

<operation body> as well as the <statement>s in <operation definition> may contain neither an <imperative expression> nor an <identifier> defined outside the enclosing <operation definition> or <operation diagram> respectively, except for <synonym identifier>s, <operation identifier>s, <literal identifier>s and <sort>s.

If an exception can be raised in an operation when no exception handler is active with the corresponding on-exception clause (that is, it is not handled), <raises> must mention this exception. An exception is considered as not handled in an operation if there is a potential control flow inside the operation producing that exception, and none of the exception handlers activated in this control flow handles the exception.

The list of <variable name>s is considered to bind tighter than the list of <operation parameters> within <formal operation parameters>.

Concrete graphical grammar

```
<operation diagram> ::=
                        <frame symbol> contains {
                                <operation heading>
                                { <operation text area>* <operation graph area> }set }
                        [ is associated with <package use area> ]

<operation graph area> ::=
                        [ <on exception association area> ] <procedure start area>
                        { <in connector area> | <exception handler area> }*

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

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

The <start> in <operation diagram> must not contain <virtuality>.

As there is no graphical grammar for sort definitions, an <operation diagram> can only be used through a <textual operation reference>.

For each <operation diagram>, there must exist an <operation signature> in the same scope unit having the same <operation name>, positionally having the same argument <sort>s as specified in the <formal operation parameters> (if present) and having the same result <sort> as specified in <operation result>. If there is only one operation with the given <operation identifier> or <operation name>, then the argument <sort>s in <formal operation parameters> may be omitted. Likewise, the <sort> may be omitted in <operation result>.

For each <operation signature> at most one corresponding <operation diagram> can be given.

<operation body> as well as the <statement>s in <operation definition> may contain neither an <imperative expression> nor an <identifier> defined outside the enclosing <operation definition> or <operation diagram> respectively, except for <synonym identifier>s, <operation identifier>s, <literal identifier>s and <sort>s.

Semantics

An operator is a constructor for elements of the sort identified by the result. It must always return either a value, or a newly constructed object. In contrast, a method may return an existing object.

An operator must not modify objects that are reachable by following references from the actual parameters or the actual parameters themselves. An object is considered modified in an operator if there is a potential control flow inside the operator resulting in that modification.

An operation definition is a scope unit defining its own data and variables that can be manipulated inside the <operation body>.

If the <operation heading> begins with the keyword **operator**, then <operation definition> defines the behaviour of an operator. If the <operation heading> begins with the keyword **method**, then <operation definition> defines the behaviour of a method.

Variables introduced in <formal operation parameters> are local variables of the operator or method, and can be modified within <operation body>.

An <external operation definition> is an operator or method whose behaviour is not included in the SDL description. Conceptually, an <external operation definition> is assumed to be given behaviour and to be transformed into an <operation definition> as part of the generic system transformation (see Annex F).

Model

For every <operation definition> which does not have a corresponding <operation signature>, an <operation signature> is constructed.

An <operation definition> is transformed into a <procedure definition> or <procedure diagram> respectively, having anonymous name, having <procedure formal parameters> derived from the <formal operation parameters>, and having a <result> derived from the <operation result>. The <procedure body> is derived from <operation body> if one was present, or, if the "<operation definition> contains a <statement list>, the result of this transformation is a <procedure definition> (see 9.4). After the *Model* of <procedure definition> has been applied, the virtual start inserted by that *Model* is replaced by a start without <virtuality>.

An <operation diagram> is transformed into a <procedure diagram> in a similar manner.

The *Procedure-definition* corresponding to the resultant <procedure definition> or <procedure diagram> is associated with the *Operation-signature* represented by the <operation signature>.

If the <operation definition> or <operation diagram> defines a method, then during the transformation into a <procedure definition>, or <procedure diagram> an initial parameter with <parameter kind> **in/out** is inserted into <formal operation parameters>, with the argument <sort> being the sort that is defined by the <data type definition> that constitutes the scope unit in which the <operation definition> occurs. The <variable name> in <formal operation parameters> for this inserted parameter is a newly formed anonymous name.

NOTE – It is not possible to specify an <operation definition> for a <literal signature>.

If any <operation definition> contains informal text, then the interpretation of expressions involving application of the corresponding operator or method is not formally defined by SDL but may be determined from the informal text by the interpreter. If informal text is specified, a complete formal specification has not been given in SDL.

12.1.9 Additional data definition constructs

This subclause introduces further constructs that may be used for data.

12.1.9.1 Name class

A name class is shorthand for writing a (possibly infinite) set of literal names or operator names defined by a regular expression.

A <name class literal> is an alternative way of specifying a <literal name>. A <name class operation> is an alternative way of specifying an <operation name> of a nullary operation.

Concrete textual grammar

```
<name class literal> ::=
                        nameclass <regular expression>

<name class operation> ::=
                        <operation name> in nameclass <regular expression>

<regular expression> ::=
                        <partial regular expression> { [or ] <partial regular expression> }*

<partial regular expression> ::=
                        <regular element> [ <Natural literal name> | <plus sign> | <asterisk> ]

<regular element> ::=
                        ( <regular expression> )
                   |    <character string>
                   |    <regular interval>

<regular interval> ::=
                        <character string> <colon> <character string>
```

The names formed by the <regular expression> must satisfy either the lexical rules for names or <character string>, <hex string>, or <bit string> (see 6.1).

The <character string>s in a <regular interval> must both have a length of one, excluding the leading and trailing <apostrophe>s.

A <name class operation> can only be used in an <operation signature>. An <operation signature> containing <name class operation> must only occur in an <operator list> and must not contain <arguments>.

When a name contained in the equivalent set of names of a <name class operation> occurs as the <operation name> in an <operation application>, it must not have <actual parameters>.

The equivalent set of names of a name class is defined as the set of names that satisfy the syntax specified by the <regular expression>. The equivalent sets of names for the <regular expression>s contained in a <data type definition> must not overlap.

Model

A <name class literal> is equivalent to this set of names in the abstract syntax. When a <name class operation> is used in an <operation signature>, a set of <operation signature>s is created by substituting each name in the equivalent set of names for the <name class operation> in the <operation signature>.

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. **or** is more tightly binding than simple sequencing.

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>.

If a <regular element> is followed by '*' the <partial regular expression> is equivalent to the <regular element> being repeated zero or more times.

If a <regular element> is followed by <plus sign> the <partial regular expression> is equivalent to the <regular element> being repeated one or more times.

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> defines the character sequence given in the character string (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 D.2).

The names generated by a <name class literal> are defined in the alphabetical order according to the ordering of the character sort. The characters are considered case sensitive, and a true prefix of a word is considered less than the whole word.

NOTE – Examples can be found in Annex D.

12.1.9.2 Name class mapping

A name class mapping is shorthand for defining a (possibly infinite) number of operation definitions ranging over all the names in a <name class operation>. The name class mapping allows behaviour to be defined for the operators and methods defined by a <name class operation>. A name class mapping takes place when an <operation name> that occurred in a <name class operation> within an <operation signature> of the enclosing <data type definition> is used in <operation definitions> or <operation diagram>s.

A spelling term in a name class mapping refers to the character string that contains the spelling of the name. This mechanism allows the Charstring operations to be used to define name class mappings.

Concrete textual grammar

```
<spelling term> ::=
                    spelling (<operation name>)
```

A <spelling term> is legal concrete syntax only within an <operation definition> or <operation diagram>, if a name class mapping has taken place.

Model

A name class mapping is shorthand for a set of <operation definition>s or a set of <operation diagram>s. The set of <operation definition>s is derived from an <operation definition> by substituting each name in the equivalent set of names of the corresponding <name class operation> for each occurrence of <operation name> in the <operation definition>. The derived set of <operation definition>s contains all possible <operation definition>s that can be generated in this way. The same procedure is followed for deriving a set of <operation diagram>s.

The derived <operation definition>s and <operation diagram>s are considered legal even though a <string name> is not allowed as an <operation name> in the concrete syntax.

The derived <operation definition>s are added to <operation definitions> (if any) in the same <data type definition>. The derived <operation diagram>s are added to the list of diagrams where the original <operation definition> had occurred.

If an <operation definition> or <operation diagram> contains one or more <spelling term>s, each <spelling term> is replaced with a Charstring literal (see D.3.4).

If, during the above transformation, the <operation name> in the <spelling term> had been replaced by an <operation name>, the <spelling term> is shorthand for a Charstring derived from the <operation name>. The Charstring contains the spelling of the <operation name>.

If, during the above transformation, the <operation name> in the <spelling term> had been replaced by a <string name>, the <spelling term> is shorthand for a Charstring derived from the <string name>. The Charstring contains the spelling of the <string name>.

12.1.9.3 Restricted visibility

Concrete textual grammar

```
<visibility> ::=
                    public | protected | private
```

<visibility> must not precede a <literal list>, <structure definition>, or <choice definition> in a <data type definition> containing <data type specialization>. <visibility> must not be used in an <operation signature> that redefines an inherited operation signature.

Semantics

<visibility> controls visibility of a literal name or operation name.

When a <literal list> is preceded by <visibility>, this <visibility> applies to all <literal signature>s. When a <structure definition> or <choice definition> is preceded by <visibility>, then this <visibility> applies to all implied <operation signatures>.

When a <fields of sort> or <choice of sort> is preceded by a <visibility>, this <visibility> applies to all implied <operation signatures>.

If a <literal signature> or <operation signature> contains the keyword **private** in <visibility>, then the *Operation-name* derived from this <operation signature> is only visible within the scope of the <data type definition> that contains the <operation signature>. When a <data type definition> containing such <operation signature> is specialized, the <operation name> in <operation signature> is implicitly renamed to an anonymous name. Every occurrence of this <operation name> within the <operation definitions> or <operation diagram>s corresponding to this <operation signature> is renamed to the same anonymous name, when the <operation signature> and the corresponding operation definition are inherited by specialization.

NOTE 1 – As a consequence, the operator or method defined by this <operation signature> can only be used in operation applications within the data type definition that originally defined this <operation signature>, but not in any subtype thereof.

If a <literal signature> or <operation signature> contains the keyword **protected** in <visibility>, then the *Operation-name* derived from this <operation signature> is only visible within the scope of the <data type definition> that contains the <operation signature>.

NOTE 2 – Because inherited operators and methods are copied into the body of the subtype, the operator or method defined by this <operation signature> can be accessed within the scope of any <data type definition> that is a subtype of the <data type definition> that originally defined this <operation signature>.

NOTE 3 – If a <literal signature> or <operation signature> does not contain <visibility>, the *Operation-name* derived from this <operation signature> is visible everywhere where the <sort name> that is defined in the enclosing <data type definition> is visible.

Model

If a <literal signature> or <operation signature> contains the keyword **public** in <visibility>, this is derived syntax for a signature having no protection.

12.1.9.4 Syntypes

A syntype specifies a subset of the elements of a sort. A syntype used as a sort has the same semantics as the sort referenced by the syntype except for checks that data items belong to the specified subset of the elements of the sort.

Abstract grammar

Syntype-identifier	=	*Identifier*
Syntype-definition	::	*Syntype-name*
		Parent-sort-identifier
		Range-condition
Syntype-name	=	*Name*
Parent-sort-identifier	=	*Sort-identifier*

Concrete textual grammar

```
<syntype> ::=
                        <syntype identifier>

<syntype definition> ::=
                        {<package use clause>}*
                        syntype <syntype name> <equals sign> <parent sort identifier>
                            [<default initialization>] [<constraint> <end>]
                            [<syntype closing> <end> ]
                    |   {<package use clause>}*
                    |   <type preamble> <data type heading> [<data type specialization>]
                        {   <constraint> <end>
                          | <end> <data type definition body> <constraint> <end> <data type closing> <end>
                          | <left curly bracket> <data type definition body> <constraint> <end>
                                <right curly bracket> }

<syntype closing> ::=
                        endsyntype [<syntype name>]

<parent sort identifier> ::=
                        <sort>
```

A <syntype> is an alternative for a <sort>.

A <syntype definition> with the keywords **value type** or **object type** 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 a <sort> in <arguments> when defining an operation, the sort for the corresponding *Formal-argument*s is the *Parent-sort-identifier* of the syntype.

When a <syntype identifier> is used as a result of an operation, 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 <constraint> is omitted, then the <syntype identifier>s for the syntype are in the Abstract grammar represented as the *Parent-sort-identifier*.

Semantics

A syntype definition defines a syntype, which references a sort identifier and a constraint. 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 12.3.3);

b) output of a signal if one of the sorts specified for the signal is a syntype (see 10.3 and 11.13.4);

c) calling a procedure when one of the sorts specified for the procedure in parameter variables is a syntype (see 9.4 and 11.13.3);

d) creating an agent when one of the sorts specified for the agent parameters is a syntype (see 9.3 and 11.13.2);

e) input of a signal and one of the variables which is associated with the input, has a sort which is a syntype (see 11.2.2);

f) calling an operation application that has a syntype defined as either an argument sort or a result sort (see 12.2.7);

g) set or reset clause or active expression on a timer and one of the sorts in the timer definition is a syntype (see 11.15 and 12.3.4.4);

h) remote variable or remote procedure definition if one of the sorts for derivation of implicit signals is a syntype (see 10.5 and 10.6);

i) procedure formal context parameter with an in/out or out parameter in <procedure signature> matched with an actual context parameter where the corresponding formal parameter or the in/out or out parameter in the <procedure signature> is a syntype;

j) <any expression>, where the result will be within the range (see 12.3.4.5);

k) raise of an exception if one of the sorts specified for the exception is a syntype (see 11.12.2.5).

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-condition* that constrains the sort. If a range condition is used, the sort is constrained to the set of data items specified by the constants of the syntype definition. If a size constraint is used, the sort is constrained to contain data items given by the size constraint.

Model

A <syntype definition> with the keywords **value type** or **object type** can be distinguished from a <data type definition> by the inclusion of a <constraint>. Such a <syntype definition> is shorthand for introducing a <data type definition> with an anonymous name followed by a <syntype definition> with the keyword **syntype** based on this anonymously named sort and including <constraint>.

12.1.9.5 Constraint

Abstract grammar

Range-condition	::	*Condition-item-**set***
Condition-item	=	*Open-range* | *Closed-range*
Open-range	::	*Operation-identifier*
		Constant-expression
Closed-range	::	*Open-range*
		Open-range

Concrete textual grammar

```
<constraint> ::=
                        constants ( <range condition> )
                    |   <size constraint>

<range condition> ::=
                        <range> { , <range> }*

<range> ::=
                        <closed range>
                    |   <open range>
```

```
<closed range> ::=
                        <constant> <colon> <constant>

<open range> ::=
                        <constant>
                  |     {     <equals sign>
                        |     <not equals sign>
                        |     <less than sign>
                        |     <greater than sign>
                        |     <less than or equals sign>
                        |     <greater than or equals sign> } <constant>

<size constraint> ::=
                        size ( <range condition> )

<constant> ::=
                        <constant expression>
```

The symbol "<" must only be used in the concrete syntax of the <range condition> if that symbol has been defined with an <operation signature>:

```
"<"  ( P, P ) -> <<package Predefined>>Boolean;
```

where P is the sort of the syntype, and similarly for the symbols ("<=", ">", ">=", respectively). These symbols represent *Operation-identifier*.

A <closed range> must only be used if the symbol "<=" is defined with an <operation signature>:

```
"<="( P, P ) -> <<package Predefined>>Boolean;
```

where P is the sort of the syntype.

A <constant expression> in a <range condition> must have the same sort as the sort of the syntype.

A <size constraint> must only be used in the concrete syntax of the <range condition> if the symbol Length has been defined with an <operation signature>:

```
Length ( P ) -> <<package Predefined>>Natural;
```

where P is the sort of the syntype.

Semantics

A constraint defines a range check. A range check is used when a syntype has additional semantics to the sort of the syntype [see 12.3.1, 12.1.9.4 and the cases where syntypes have different semantics – see the subclauses referenced in items a) to k) in 12.1.9.4, *Semantics*]. A range check is also used to determine the interpretation of a decision (see 11.13.5).

The range check is the application of the operation formed from the range condition or size constraint. For syntype range checks, the application of this operation must be equivalent to the predefined Boolean value true otherwise the predefined exception OutOfRange (see D.3.16) is raised. The range check is derived as follows:

a) Each <open range> or <closed range> in the <range condition> has a corresponding *Open-range* (predefined Boolean **or**) or *Closed-range* (predefined Boolean **and)** in the *Condition-item*.

b) An <open range> of the form <constant> is equivalent to an <open range> of the form = <constant>.

c) For a given expression, A, then:

   1) an <open range> of the form = <constant>, /= <constant>, < <constant>, <less than or equals sign> <constant>, > <constant>, and <greater than or equals sign> <constant>, has sub-expression in the range check of the form A = <constant>, A /= <constant>, A <

<constant>, A <less than or equals sign> <constant>, A > <constant>, and A <greater than or equals sign> <constant> respectively;

2) a <closed range> of the form *first* <constant> : *second* <constant> has a sub-expression in the range check of the form *first* <constant> <less than or equals sign> A **and** A <less than or equals sign> *second* <constant> where **and** corresponds to the predefined Boolean **and**;

3) a <size constraint> has a sub-expression in the range check of the form Length(A) = <range condition>.

d) There is a predefined Boolean **or** operation for the distributed operation over all the data items in the *Condition-item*-**set**. The range check is the expression formed from the predefined Boolean **or** of all the sub-terms derived from the <range condition>.

If a syntype is specified without a <constraint> then the range check is the predefined Boolean value true.

12.1.9.6 Synonym definition

A synonym gives a name to a constant expression that represents one of the data items of a sort.

Concrete textual grammar

```
<synonym definition> ::=
                        synonym <synonym definition item> { , <synonym definition item> }*<end>

<synonym definition item> ::=
                        <internal synonym definition item>
                    |   <external synonym definition item>

<internal synonym definition item> ::=
                        <synonym name> [<sort>] <equals sign> <constant expression>

<external synonym definition item> ::=
                        <synonym name> <predefined sort> = external
```

The <constant expression> in the concrete syntax denotes a *Constant-expression* in the abstract syntax as defined in 12.2.1.

If a <sort> is specified, the result of the <constant expression> has a static sort of <sort>. It must be possible for <constant expression> to have that sort.

If the sort of the <constant_expression> cannot be uniquely determined, then a sort must be specified in the <synonym definition>.

The <constant expression> must not refer to the synonym defined by the <synonym definition> either directly or indirectly (via another synonym).

An <external synonym definition item> defines a <synonym> whose result is not defined in a specification. Conceptually, an <external synonym definition item> is assumed to be given a result and to be transformed into <internal synonym definition item> as part of the generic system transformation (see Annex F).

Semantics

The result that the synonym represents is determined by the context in which the synonym definition appears.

If the sort of the constant expression cannot be uniquely determined in the context of the synonym, then the sort is given by the <sort>.

A synonym has a result, which is the result of the constant expression in the synonym definition.

A synonym has a sort, which is the sort of the constant expression in the synonym definition.

12.2 Passive use of data

The following subclauses define how sorts, literals, operators, methods and synonyms are interpreted in expressions.

12.2.1 Expressions

Abstract grammar

```
Expression                    =   Constant-expression
                              |   Active-expression
Constant-expression           =   Literal
                              |   Conditional-expression
                              |   Equality-expression
                              |   Operation-application
                              |   Range-check-expression
Active-expression             =   Variable-access
                              |   Conditional-expression
                              |   Operation-application
                              |   Equality-expression
                              |   Imperative-expression
                              |   Range-check-expression
                              |   Value-returning-call-node
```

Concrete textual grammar

For simplicity of description, no distinction is made between the concrete syntax of *Constant-expression* and *Active-expression.*

```
<expression> ::=
                    <operand>
              |     <create expression>
              |     <value returning procedure call>

<operand> ::=
                    <operand0>
              |     <operand> <implies sign> <operand0>

<operand0> ::=
                    <operand1>
              |     <operand0> { or | xor } <operand1>

<operand1> ::=
                    <operand2>
              |     <operand1> and <operand2>

<operand2> ::=
                    <operand3>
              |     <operand2> { <greater than sign>
                               | <greater than or equals sign>
                               | <less than sign>
                               | <less than or equals sign>
                               | in } <operand3>
              |     <range check expression>
              |     <equality expression>

<operand3> ::=
                    <operand4>
              |     <operand3> { <plus sign> | <hyphen> | <concatenation sign> } <operand4>

<operand4> ::=
                    <operand5>
              |     <operand4> { <asterisk> | <solidus> | mod | rem } <operand5>

<operand5> ::=
                    [ <hyphen> | not ] <primary>
```

```
<primary> ::=
                        <operation application>
                |       <literal>
                |       ( <expression> )
                |       <conditional expression>
                |       <spelling term>
                |       <extended primary>
                |       <active primary>
                |       <synonym>

<active primary> ::=
                        <variable access>
                |       <imperative expression>

<expression list> ::=
                        <expression> { , <expression> }*

<simple expression> ::=
                        <constant expression>

<constant expression> ::=
                        <constant expression>
```

An <expression> that does not contain any <active primary>, a <create expression>, or a <value returning procedure call> with a <remote procedure call body> is a <constant expression>. A <constant expression> represents a *Constant-expression* in the abstract syntax. Each <constant expression> is interpreted once during initialization of the system, and the result of the interpretation is preserved. Whenever the value of the <constant expression> is needed during interpretation, a complete copy of that computed value is used.

An <expression> that is not a <constant expression> represents an *Active-expression*.

If an <expression> contains an <extended primary>, the <extended primary> is replaced at the concrete syntax level as defined in 12.2.4 before relationship to the abstract syntax is considered.

<operand>, <operand1>, <operand2>, <operand3>, <operand4> and <operand5> offer special syntactic forms for operator and method names. The special syntax is introduced, for example, so that arithmetic operations and Boolean operations 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 operation will depend on the data type definition.

A <simple expression> must contain only literals, operators, and methods defined within the package Predefined, as defined in Annex D.

Semantics

An infix operator or method in an expression has the normal semantics of an operator or method but with infix or quoted prefix syntax.

A monadic operator or method in an expression has the normal semantics of an operator or method but with the prefix or quoted prefix syntax.

Infix operators or methods have an order of precedence that determines the binding of operators or methods. When the binding is ambiguous, then binding is from left to right.

When an expression is interpreted it returns a data item (a value, object or pid). The returned data item is referred to as the result of the expression.

The (static) sort of an expression is the sort of the data item that would be returned by the interpretation of the expression as determined from analysis of the specification without consideration of the interpretation semantics. The dynamic sort of an expression is the sort of the result of the expression. The static and dynamic sort of active expressions may differ due to polymorphic assignments (see 12.3.3). For a constant expression, the dynamic sort of an expression is its static sort.

NOTE – To avoid cumbersome text, the word "sort" always refers to a static sort, unless preceded by the word "dynamic". For clarity "static sort" is written explicitly in some cases.

A simple expression is a *Constant-expression*.

Model

An expression of the form:

    <expression> <infix operation name> <expression>

is derived syntax for:

    <quotation mark> <infix operation name> <quotation mark> **(** <expression>**,** <expression> **)**

where <quotation mark> <infix operation name> <quotation mark> represents an *Operation-name*.

Similarly,

    <monadic operation name> <expression>

is derived syntax for:

    <quotation mark> <monadic operation name> <quotation mark> **(** <expression> **)**

where <quotation mark> <monadic operation name> <quotation mark> represents an *Operation-name*.

12.2.2 Literal

Abstract grammar

Literal	::	*Literal-identifier*
Literal-identifier	=	*Identifier*

The *Literal-identifier* denotes a *Literal-signature*.

Concrete textual grammar

```
<literal> ::=
                        <literal identifier>

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

Whenever a <literal identifier> is specified, the unique *Literal-name* in *Literal-identifier* is derived in the same way, with the result sort derived from context. A *Literal-identifier* is derived from context (see 6.3) so that if the <literal identifier> is overloaded (that is, the same name is used for more than one literal or operation) then the *Literal-name* identifies a visible literal with the same name and result sort consistent with the literal. Two literals with the same <name> but differing by result sorts have different *Literal-names*.

It must be possible to bind each unqualified <literal identifier> to exactly one defined *Literal-identifier* which satisfies the conditions in the construct in which the <literal identifier> is used.

Wherever a <qualifier> of a <literal identifier> contains a <path item> with the keyword **type**, then the <<u>sort</u> name> after this keyword does not form part of the *Qualifier* of the *Literal-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

A *Literal* returns the unique data item corresponding to its *Literal-signature*.

The sort of the <literal> is the *Result* in its *Literal-signature*.

12.2.3 Synonym

Concrete textual grammar

```
<synonym> ::=
                          <synonym identifier>
```

Semantics

A synonym is shorthand for denoting an expression defined elsewhere.

Model

A <synonym> represents the <constant expression> defined by the <synonym definition> identified by the <synonym identifier>. An <identifier> used in the <constant expression> represents an *Identifier* in the abstract syntax according to the context of the <synonym definition>.

12.2.4 Extended primary

An extended primary is a shorthand syntactic notation. However, apart from the special syntactic form, an extended primary has no special properties and denotes an operation and its parameter(s).

Concrete textual grammar

```
<extended primary> ::=
                          <indexed primary>
                    |     <field primary>
                    |     <composite primary>

<indexed primary> ::=
                          <primary> ( <actual parameter list> )
                    |     <primary> <left square bracket> <actual parameter list> <right square bracket>

<field primary> ::=
                          <primary> <exclamation mark> <field name>
                    |     <primary> <full stop> <field name>
                    |     <field name>

<field name> ::=
                          <name>

<composite primary> ::=
                          [<qualifier>]  <composite begin sign> <actual parameter list> <composite end sign>
```

Model

An <indexed primary> is derived concrete syntax for:

<primary> <full stop> Extract **(** <actual parameter list> **)**

The abstract syntax is determined from this concrete expression according to 12.2.1.

A <field primary> is derived concrete syntax for:

<primary> <full stop> *field-extract-operation-name*

where the *field-extract-operation-name* is formed from the concatenation of the field name and "Extract" in that order. The abstract syntax is determined from this concrete expression according to 12.2.1. The transformation according to this model is performed before the modification of the signature of methods in 12.1.4.

When the <field primary> has the form <field name>, this is derived syntax for:

this ! <field name>

A <composite primary> is derived concrete syntax for:

<qualifier> Make **(** <actual parameter list> **)**

if any actual parameters were present, or:

<qualifier> Make

otherwise, and where the <qualifier> is inserted only if it was present in the <composite primary>. The abstract syntax is determined from this concrete expression according to 12.2.1.

12.2.5 Equality expression

Abstract grammar

Equality-expression	::	*First-operand*
		Second-operand
First-operand	=	*Expression*
Second-operand	=	*Expression*

An *Equality-expression* represents the equality of either references or values of its *First-operand* and its *Second-operand*.

Concrete textual grammar

```
<equality expression> ::=
                    <operand2> { <equals sign> | <not equals sign> } <operand3>
```

An <equality expression> is legal concrete syntax only if the sort of one of its operands is sort compatible to the sort of the other operand.

Semantics

Interpretation of the *Equality-expression* proceeds by interpretation of its *First-operand* and its *Second-operand*.

If, after interpretation, both operands are objects, then the *Equality-expression* denotes reference equality. It returns the predefined Boolean value true if and only if both operands are either Null or reference the same object data item.

If, after interpretation, both operands are pids, then the *Equality-expression* denotes agent identity. It returns the predefined Boolean value true if and only if both operands are either Null or reference the same agent instance.

If, after interpretation, either one of the operands is a value, the *Equality-expression* denotes equality of values as follows:

a) If the dynamic sort of *First-operand* is sort compatible to the dynamic sort of *Second-operand*, the <equality expression> returns the result of the application of the equal operator to *First-operand* and *Second-operand*, where equal is the *Operation-identifier* represented by the <operation identifier> in the <operation application>:

   equal(<operand2>, <operand3>)

b) Otherwise, the <equality expression> returns the result of the application of the equal operator to *Second-operand* and *First-operand*, where equal is the *Operation-identifier* represented by the <operation identifier> in the <operation application>:

   equal(<operand3>, <operand2>)

The concrete syntax form:

<operand2> <not equals sign> <operand3>

is derived concrete syntax for:

not (<operand2> = <operand3> **)**

where **not** is an operation of the predefined Boolean data type.

12.2.6 Conditional expression

Abstract grammar

Conditional-expression	::	*Boolean-expression*
		Consequence-expression
		Alternative-expression
Boolean-expression	=	*Expression*
Consequence-expression	=	*Expression*
Alternative-expression	=	*Expression*

A *Conditional-expression* is an *Expression,* which is interpreted as either the *Consequence-expression* or the *Alternative-expression*.

The sort of the *Consequence-expression* must be the same as the sort of the *Alternative-expression*.

Concrete textual grammar

```
<conditional expression> ::=
                    if <Boolean expression>
                    then <consequence expression>
                    else <alternative expression>
                    fi

<consequence expression> ::=
                    <expression>

<alternative expression> ::=
                    <expression>
```

The sort of the <consequence expression> must be the same as the sort of the <alternative expression>.

Semantics

A conditional expression represents an *Expression* that is interpreted as either the *Consequence-expression* or the *Alternative-expression*.

If the *Boolean-expression* returns the predefined Boolean value true then the *Alternative-expression* is not interpreted. If the *Boolean-expression* returns the predefined Boolean value false then the *Consequence-expression* is not interpreted.

A conditional expression has a sort, which is the sort of the consequence expression (and also the sort of the alternative expression).

The result of the conditional expression is the result of interpreting the *Consequence-expression* or the *Alternative-expression*.

The static sort of a conditional expression is the static sort of the *Consequence-expression* (which is also the sort of the *Alternative-expression*). The dynamic sort of the conditional expression is the dynamic sort of the result of interpreting the conditional expression.

12.2.7 Operation application

Abstract grammar

Operation-application	::	*Operation-identifier*
		[*Expression*]*
Operation-identifier	=	*Identifier*

The *Operation-identifier* denotes an *Operation-signature*, either a *Static-operation-signature* or a *Dynamic-operation-signature*. Each *Expression* in the list of *Expression*s after the *Operation-identifier* must be sort compatible to the corresponding (by position) sort in the list of *Formal-argument*s of the *Operation-signature*.

Each *Operation-signature* has associated a *Procedure-definition*, as described in 12.1.8.

Each *Expression* corresponding by position to an *Inout-parameter* or *Out-parameter* in the *Procedure-definition* associated with the *Operation-signature* must be a *Variable-identifier* having the same *Sort-reference-identifier* as the corresponding (by position) sort in the list of *Formal-argument*s of the *Operation-signature*.

Concrete textual grammar

```
<operation application> ::=
                                <operator application>
                        |       <method application>

<operator application> ::=
                                <operation identifier> [<actual parameters>]

<method application> ::=
                                <primary> <full stop> <operation identifier> [<actual parameters>]
```

Whenever an <operation identifier> is specified, the unique *Operation-name* in *Operation-identifier* is derived in the same way. The list of argument sorts is derived from the actual parameters and the result sort is derived from context (see 6.3). Therefore, if the <operation name> is overloaded (that is, the same name is used for more than one literal or operation), the *Operation-name* identifies a visible operation with the same name and the argument sorts and result sort consistent with the operation application. Two operations with the same <name> but differing by one or more of the argument or result sorts have different *Operation-names*.

It must be possible to bind each unqualified <operation identifier> to exactly one defined *Operation-identifier* which satisfies the conditions in the construct in which the <operation identifier> is used.

When the operation application has the syntactical form:

    <operation identifier> [<actual parameters>]

then, during derivation of the *Operation-identifier* from context, the form:

    this <full stop> <operation identifier> [<actual parameters>]

is also considered. The model in 12.3.1 is applied before resolution by context is attempted.

Wherever a <qualifier> of an <operation 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 *Operation-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**.

If all the <expression>s in the parenthesized list of <expression>s are <constant expression>s, the <operation application> represents a *Constant-expression* as defined in 12.2.1.

A <method application> is legal concrete syntax only if <operation identifier> represents a method.

An <expression> in <actual parameters> corresponding to an *Inout-parameter* or *Out-parameter* in the *Procedure-definition* associated with the *Operation-signature* cannot be omitted and must be a <variable access> or <extended primary>.

If **this** is used, <operation identifier> must denote an enclosing operator or method.

NOTE – <actual parameters> may be omitted in an <operation application> if all actual parameters have been omitted.

Semantics

Resolution by context (see 6.3) guarantees that an operation is selected such that the types of the actual arguments are pairwise sort compatible to the types of the formal arguments.

An operation application with an *Operation-identifier* that denotes a *Static-operation-signature* is interpreted by transferring the interpretation to the *Procedure-definition* associated with the *Operation-signature* and that procedure graph is interpreted (the explanation is contained in 9.4).

An operation application with an *Operation-identifier* that denotes a *Dynamic-operation-signature*, is interpreted by the following steps:

a) the actual parameters are interpreted;

b) if the result of an actual parameter corresponding to a *Virtual-argument* was Null, predefined exception InvalidReference is raised;

c) all *Dynamic-operation-signatures* are collected into a set such that the operation signature formed from an *Operation-name* derived from the <operation name> in <operation identifier> and the dynamic sorts of the result of interpreting the actual parameters is sort compatible to the candidate *Dynamic-operation-signature*;

d) the unique *Dynamic-operation-signature* that is sort compatible to all other *Dynamic-operation-signatures* in this set is selected; and

e) the interpretation is transferred to the *Procedure-definition* associated with the selected *Operation-signature* and that procedure graph is interpreted (the explanation is contained in 9.4).

The existence of such a unique signature is guaranteed by the requirement that the set of *Dynamic-operation-signatures* form a lattice (see 12.1.2).

The list of actual parameter *Expression*s in an *Operation-application* is interpreted in the order given from left to right before the operation itself is interpreted.

If an <expression> in <actual parameters> is omitted, the corresponding formal parameter has no data item associated; that is, it is "undefined".

If an argument sort of the operation signature is a syntype, then the range check defined in 12.1.9.5 is applied to the result of the *Expression*. If the range check is the predefined Boolean value false at the time of interpretation, then the predefined exception OutOfRange (see D.3.16) is raised.

The interpretation of the transition containing the <operation application> continues when the interpretation of the called procedure is finished. The result of the operation application is the result returned by the interpretation of the referenced procedure definition.

If the result sort of the operation signature is a syntype then the range check defined in 12.1.9.5 is applied to the result of the operation application. If the range check is the predefined Boolean value false at the time of interpretation, then the predefined exception OutOfRange (see D.3.16) is raised.

An <operation application> has a sort, which is the sort of the result obtained by the interpretation of the associated procedure.

Model

The concrete syntax form:

```
<expression> <full stop> <operation identifier> [<actual parameters>]
```

is derived concrete syntax for:

<operation identifier> *new-actual-parameters*

where *new-actual-parameters* is <actual parameters> containing only <expression>, if <actual parameters> was not present; otherwise *new-actual-parameters* is obtained by inserting <expression> before the first optional expression in <actual parameters>.

12.2.8 Range check expression

Abstract grammar

Range-check-expression :: *Range-condition Expression*

Concrete textual grammar

<range check expression> ::=
  <operand2> **in type** { <<u>sort</u> identifier> <constraint> | <sort> }

The sort of <operand2> must be the same as the sort identified by <<u>sort</u> identifier> or <sort>.

Semantics

A *Range-Check-Expression* is an expression of the predefined Boolean sort which has the result true if the result of the *Expression* fulfils the *Range-condition* corresponding to <constraint> as defined in 12.1.9.5, otherwise it has the result false.

Model

Specifying a <sort> is derived syntax for specifying the <constraint> of the data type that defined the <sort>. If that data type was not defined with a <constraint>, the <range check expression> is not evaluated and the <range check expression> is derived syntax for specifying the predefined Boolean value true.

12.3 Active use of data

This subclause defines the use of data and declared variables, how an expression involving variables is interpreted, and the imperative expressions which obtain results from the underlying system.

A variable has a sort and an associated data item of that sort. The data item associated with a variable may be changed by assigning a new data item to the variable. The data item associated with the variable may be used in an expression by accessing the variable.

Any expression containing a variable is considered to be "active", because the data item obtained by interpreting the expression may vary according to the data item last assigned to the variable. The result of interpreting an active expression will depend on the current state of the system.

12.3.1 Variable definition

A variable has a data item associated, or it is "undefined".

Abstract grammar

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

If the *Constant-expression* is present, it must be of the same sort as the *Sort-reference-identifier* denoted.

Concrete textual grammar

<variable definition> ::=
  dcl [**exported**] <variables of sort> {**,** <variables of sort> }* <end>

<variables of sort> ::=
  <<u>variable</u> name> [<exported as>] { **,** <<u>variable</u> name> [<exported as>] }*
    <sort> [<is assigned sign> <constant expression>]

<exported as> ::=
  as <<u>remote variable</u> identifier>

<exported as> may only be used for a variable with **exported** in its <variable definition>. Two exported variables in an agent cannot mention the same <remote variable identifier>.

The *Constant-expression* is represented by:

a) if a <constant expression> is given in the <variable definition>, then this <constant expression>;

b) else, if the data type that defined the <sort> has a <default initialization>, then the <constant expression> of the <default initialization>.

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

Semantics

When a variable is created and the *Constant-expression* is present, then the variable is associated with the result of the *Constant-expression*. Otherwise, the variable has no data item associated: that is, the variable is "undefined".

If *Sort-reference-identifier* is a *Syntype-identifier*, *Constant-expression* is present and the result of the *Constant-expression* does not comply with the *Range-condition*, the predefined exception OutOfRange is raised (see D.3.16).

The keyword **exported** allows a variable to be used as an exported variable as elaborated in 10.6.

12.3.2 Variable access

Abstract grammar

```
Variable-access                =    Variable-identifier
```

Concrete textual grammar

```
<variable access> ::=
                        <variable identifier>
                   |    this
```

this must only occur in method definitions.

Semantics

A variable access is interpreted as giving the data item associated with the identified variable.

A variable access has a static sort, which is the sort of the variable identified by the variable access. It has a dynamic sort, which is the dynamic sort of the data item associated with the identified variable.

A variable access has a result, which is the data item last associated with the variable. If the variable is "undefined", a Raise-node for the predefined exception UndefinedVariable (see D.3.16) is interpreted.

Model

A <variable access> using the keyword **this** is replaced by the anonymous name introduced as the name of the leading parameter in <arguments> according to 12.1.8.

12.3.3 Assignment and assignment attempt

An assignment creates an association from the variable to the result of interpreting an expression. In an assignment attempt, this association is created only if the dynamic sorts of the variable and the expression are compatible.

Abstract grammar

Assignment	::	*Variable-identifier* *Expression*
Assignment-attempt	::	*Variable-identifier* *Expression*

In an *Assignment*, the sort of the *Expression* must be sort compatible to the sort of the *Variable-identifier*.

In an *Assignment-attempt*, the sort of the *Variable-identifier* must be sort compatible to the sort of the *Expression*.

If the variable is declared with a *Syntype* and the *Expression* is a *Constant-expression*, then the range check defined in 12.1.9.5 above applied to the *Expression* must be the predefined Boolean value true.

Concrete textual grammar

```
<assignment> ::=
                    <variable> <is assigned sign> <expression>

<variable> ::=
                    <variable identifier>
               |    <extended variable>
```

If the <variable> is a <variable identifier> then the <expression> in the concrete syntax represents the *Expression* in the abstract syntax. An <extended variable> is derived syntax and is replaced at the concrete syntax level as defined in 12.3.3.1 before relationship to the abstract syntax is considered.

If the <variable identifier> has been declared with an object sort and the sort of the <expression> is a (direct or indirect) supersort of the sort of the <variable identifier>, the <assignment> represents an *Assignment-attempt*. Otherwise, the <assignment> represents an *Assignment*.

Semantics

An *Assignment* is interpreted as creating an association from the variable identified in the assignment to the result of the expression in the assignment. The previous association of the variable is lost.

The manner in which this association is established depends on the sort of the <variable identifier> and the sort of the <expression>:

a) If the <variable identifier> has a value sort, then the result of the *Expression* is copied onto the value currently associated with *Variable-identifier* by interpreting the *copy* method defined by the data type definition that introduced the sort of the <variable identifier>, given *Variable-identifier* and *Expression* as actual parameters. If the *Expression* is Null, the predefined exception InvalidReference (see D.3.16) is raised.

b) If the <variable identifier> has an object sort and the result of the *Expression* is an object, the *Variable-identifier* is associated with the object that is the result of *Expression*. It is not allowed that the sort of the <expression> is a syntype that restricts the elements of the sort of the <variable identifier>.

c) If the <variable identifier> has an object sort and the result of the *Expression* is a value, a clone of the result of *Expression* is constructed by interpreting the *clone* operator defined by the data type definition that introduced the sort of the <variable identifier>, given *Expression* as actual parameter. The *Variable-identifier* is associated with a reference to the cloned value. It is not allowed that the sort of the <expression> is a syntype that restricts the elements of the sort of the <variable identifier>.

d) If the <variable identifier> has a pid sort and the result of the *Expression* is a pid, the *Variable-identifier* is associated with the pid that is the result of *Expression*.

If the variable is declared with a syntype, the range check defined in 12.1.9.5 is applied to the expression. If this range check returns the predefined Boolean value false, then the predefined exception OutOfRange (see D.3.16) is raised.

When an *Assignment-attempt* is interpreted, if the dynamic sort of the *Expression* is sort compatible to the sort of the *Variable-identifier*, an Assignment involving the *Variable-identifier* and the *Expression* is interpreted. Otherwise, the *Variable-identifier* is associated with Null.

NOTE – Through an assignment attempt, it is possible to determine the dynamic sort of an *Expression*.

12.3.3.1 Extended variable

An extended variable is a shorthand syntactic notation; however, apart from the special syntactic form an extended variable has no special properties and denotes an operation and its parameters.

Concrete textual grammar

```
<extended variable> ::=
                    <indexed variable>
            |       <field variable>

<indexed variable> ::=
                    <variable> ( <actual parameter list> )
            |       <variable> <left square bracket> <actual parameter list> <right square bracket>

<field variable> ::=
                    <variable> <exclamation mark> <field name>
            |       <variable> <full stop> <field name>
```

Model

<indexed variable> is derived concrete syntax for:

<variable> <is assigned sign> <variable> <full stop> Modify **(** *expressionlist* **)**

where *expressionlist* is constructed by appending <expression> to the <actual parameter list>. The abstract grammar is determined from this concrete expression according to 12.2.1. The same model applies to the second form of <indexed variable>.

The concrete syntax form:

<variable> <exclamation mark> <field name> <is assigned sign> <expression>

is derived concrete syntax for:

<variable> <full stop> *field-modify-operation-name* **(** <expression> **)**

where the *field-modify-operation-name* is formed from the concatenation of the field name and "Modify". The abstract syntax is determined from this concrete expression according to 12.2.1. The same model applies to the second form of <field variable>.

12.3.3.2 Default initialization

A default initialization allows initialization of all variables of a specified sort with the same data item, when the variables are created.

Concrete textual grammar

```
<default initialization> ::=
                    default [<virtuality>] [<constant expression>] [<end>]
```

A <data type definition> or <syntype definition> must contain at most one <default initialization>.

The <constant expression> may only be omitted if <virtuality> is **redefined** or **finalized**.

Semantics

A default initialization may be added to the <operations> of a data type definition. A default initialization specifies that any variable declared with the sort introduced by the data type definition or syntype definition initially is associated with the result of the <constant expression>.

Model

A default initialization is shorthand for specifying an explicit initialization for all those variables that are declared to be of <sort>, but where the <variable definition> was not given a <constant expression>.

If no <default initialization> is given in <syntype definition>, then the syntype has the <default initialization> of the <parent sort identifier> provided its result is in the range.

Any sort that is defined by an <object data type definition> is implicitly given a <default initialization> of Null, unless an explicit <default initialization> was present in the <object data type definition>.

Any pid sort is treated as if implicitly given a <default initialization> of Null.

If the <constant expression> is omitted in a redefined default initialization, the explicit initialization is not added.

12.3.4 Imperative expressions

Imperative expressions obtain results from the underlying system state.

The transformations described in the *Models* of this subclause are made at the same time as the expansion for import is made. A label attached to an action in which an imperative expression appears is moved to the first task inserted during the described transformation. If several imperative expressions appear in an expression, the tasks are inserted in the same order as the imperative expressions appear in the expression.

Abstract grammar

```
Imperative-expression          =    Now-expression
                               |    Pid-expression
                               |    Timer-active-expression
                               |    Any-expression
```

Concrete textual grammar

```
<imperative expression> ::=
                          <now expression>
                    |     <import expression>
                    |     <pid expression>
                    |     <timer active expression>
                    |     <any expression>
                    |     <state expression>
```

Imperative expressions are expressions for accessing the system clock, the result of imported variables, the pid associated with an agent, the status of timers or for supplying unspecified data items.

12.3.4.1 Now expression

Abstract grammar

```
Now-expression                 ::   ( )
```

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 give the same value is system dependent. However, it always holds that:

```
now <= now;
```

A now expression has the Time sort.

12.3.4.2 Import expression

Concrete textual grammar

The concrete syntax for an import expression is defined in 10.6.

Semantics

In addition to the semantics defined in 10.6, an import expression is interpreted as a variable access (see 12.3.2) to the implicit variable for the import expression.

Model

The import expression has implied syntax for the importing of the result as defined in 10.6 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 shorthand for inserting a task just before the action, where the expression occurs which assigns an implicit variable the result of the <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.

12.3.4.3 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

<create expression> ::=
                    <create request>
```

Semantics

A pid expression accesses one of the implicit anonymous variables self, parent, offspring, or sender (see clause 9, *Model*). The **self**, **parent**, **offspring** or **sender** pid expression has a result, which is the last pid associated with the corresponding implicit variable as defined in clause 9.

The dynamic sort of a <pid expression> is the dynamic sort of its result.

A **parent**, **offspring**, or **sender** pid expression has a static sort, which is Pid.

If a <create expression> includes an <agent identifier>, it has a static sort, which is the pid sort of the agent denoted by <agent identifier>. If a <create expression> includes an <agent type identifier>, it has a static sort, which is the pid sort of the agent type identified by the <agent type identifier>. If the <create expression> includes **this**, it has a static sort, which is the pid sort of the agent or agent type in which the create expression occurs. If the <create expression> includes **this** and it occurs in a context that is not inside an agent or agent type (for example in a global procedure), then it has the static sort Pid.

A **self** expression has a static sort, which is the pid of the agent or agent type in which the **self** expression occurs. If it occurs in a context that is not inside an agent or agent type (for example in a global procedure), it has the static sort Pid.

Model

The use of <create expression> in an expression is a shorthand for inserting a create request just before the action where the <create expression> occurs followed by an assignment of **offspring** to an implicitly declared anonymous variable of the same sort as the static sort of the <create expression>. The implicit variable is and then used in the expression. If <create expression> occurs several times in an expression, one distinct variable is used for each occurrence. In this case the order of the inserted create requests and variable assignments is the same as the order of the <create expression>s.

If the <create expression> contains an <agent type identifier> then the transformations that are applied to a create statement that contains an <agent type identifier> are also applied to the implicit create statements resulting from the transformation of a <create expression> (see 11.13.2).

12.3.4.4 Timer active expression

Abstract grammar

Timer-active-expression	::	*Timer-identifier*
		*Expression**

The sorts of the *Expression* list in the *Timer-active-expression* must correspond by position to the *Sort-reference-identifier* list directly following the *Timer-name* (11.15) 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 result true, if the timer identified by timer identifier and set with the same results as denoted by the expression list (if any) is active (see 11.15). Otherwise, the timer active expression has the result 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 12.1.9.5 applied to the corresponding expression in <expression list> must be the predefined Boolean value true, otherwise the predefined exception OutOfRange (see D.3.16) is raised.

12.3.4.5 Any expression

Any-expression is useful for modelling behaviour, where stating a specific data item would imply over-specification. From a result returned by an *Any-expression,* no assumption can be made on other results returned by the interpretation of *Any-expression*.

Abstract grammar

Any-expression :: *Sort-reference-identifier*

Concrete textual grammar

```
<any expression> ::=
                              any ( <sort> )
```

The <sort> must contain elements.

Semantics

An *Any-expression* returns an unspecified element of the sort or syntype designated by *Sort-reference-identifier*, if that sort or syntype is a value sort. If *Sort-reference-identifier* denotes *a Syntype-identifier*, the result will be within the range of that syntype. If the sort or syntype designated by *Sort-reference-identifier* is an object sort or pid sort, the *Any-expression* returns Null.

12.3.4.6 State expression

Concrete textual grammar

```
<state expression> ::=
                              state
```

Semantics

A state expression denotes the Character string value containing the spelling of the most recently entered state.

Model

<state expression> is derived syntax for a Charstring literal that contains the spelling of the name of the most recently entered state of the nearest enclosing scope unit. If there is no such state, <state expression> denotes the empty string (`''`). The construct is transformed together with <dash nextstate> (see Annex F).

12.3.5 Value returning procedure call

The abstract grammar for a value returning procedure call and static semantic constraints are shown in 11.13.3.

Concrete textual grammar

```
<value returning procedure call> ::=
                              [ call ] <procedure call body>
                        |     [ call ] <remote procedure call body>
```

The keyword **call** cannot be omitted if the <value returning procedure call> is syntactically ambiguous with an operation (or variable) with the same name followed by a parameter list.

NOTE 1 – This ambiguity is not resolved by context.

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 having a <procedure result>.

An <expression> in <actual parameters> corresponding to a formal **in**/**out** or **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 that contains a start transition.

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

The <procedure call body> represents a *Value-returning-call-node*, where *Procedure-identifier* is represented by the <procedure identifier>, and the list of *Expressions* is represented by the list of actual parameters. The <remote procedure call body> represents a *Value-returning-call-node*, where *Procedure-identifier* contains only the *Procedure-identifier* of the procedure implicitly defined by the *Model* below. The semantics of the *Value-returning-call-node* is shown in 11.13.3.

Model

If the <procedure identifier> is not defined within the enclosing agent, 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.

When the <value returning procedure call> contains a <remote procedure call body>, the following procedure with an anonymous name referred to as *RPCcall* is implicitly defined. *RPCsort* is the <sort> in <procedure result> of the procedure definition denoted by the <procedure identifier>.

```
procedure RPCcall -> RPCsort;
    start ;
    return call <remote procedure call body>;
endprocedure ;
```

NOTE 2 – This transformation is not again applied to the implicit procedure definition.

13 Generic system definition

A system specification may have optional parts and system parameters with unspecified results 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 data item for each of the system parameters. The resulting system specification does not contain external synonyms, and is called a specific system specification.

A generic system definition is a system definition that contains a synonym defined by an <external synonym definition item> (see 12.1.9.6), an operation defined by an <external operation definition> (see 12.1.8), a procedure defined by an <external procedure definition> (see 9.4), or <informal text> in a transition option (see 13.2). A specific system definition is created from a generic system definition by providing results for the <external synonym definition item>s, as providing behaviour for <external operation definition>s and <external procedure definition>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.

13.1 Optional definition

Concrete textual grammar

```
<select definition> ::=
                    select if ( <Boolean simple expression> ) <end>
                        {   <agent type definition>
                        |   <agent type reference>
                        |   <agent definition>
                        |   <channel definition>
                        |   <signal definition>
                        |   <signal list definition>
                        |   <remote variable definition>
                        |   <remote procedure definition>
                        |   <data definition>
                        |   <data type reference>
                        |   <composite state type definition>
                        |   <composite state type reference>
                        |   <state partitioning>
                        |   <textual typebased state partition definition>
                        |   <timer definition>
                        |   <variable definition>
                        |   <procedure definition>
                        |   <procedure reference>
                        |   <channel to channel connection>
                        |   <select definition>
                        |   <macro definition>
                        |   <exception definition> }+
                    endselect <end>
```

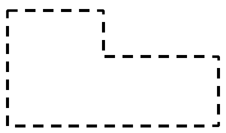
The only visible names in a <Boolean simple expression> of a <select definition> are names of external synonyms defined outside of any <select definition>s or <option area>s and literals and operations of the data types defined within the package Predefined as defined in Annex D.

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

Concrete graphical grammar

```
<option area> ::=
                    <option symbol> contains
                    { select if ( <Boolean simple expression> )
                              {   <agent type diagram>
                              |   <agent type reference area>
                              |   <agent area>
                              |   <channel definition area>
                              |   <agent text area>
                              |   <procedure text area>
                              |   <composite state type diagram>
                              |   <composite state type reference area>
                              |   <state partition area>
                              |   <procedure area>
                              |   <create line area>
                              |   <option area> } + }

<option symbol> ::=
                    { <dashed line symbol> }-set

<dashed line symbol> ::=
                    ----------------
```

The <option symbol> must form 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 (that is, with one end point inside).

An <option area> may appear anywhere, except within <state machine graph area> and <type state machine graph area>. An <option area> may contain only those areas and diagrams that are syntactically allowed at that place.

Semantics

If the result of the <Boolean simple expression> is the predefined Boolean value false, 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.

13.2 Optional transition string

Concrete textual grammar

```
<transition option> ::=
                        alternative <alternative question> <end>
                              <decision body>
                        endalternative

<alternative question> ::=
                        <simple expression>
                   |    <informal text>
```

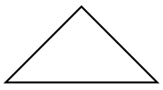
Every <constant expression> in <answer> of <decision body> must be a <simple expression>. The <answer>s in the <decision body> in a <transition option> must be mutually exclusive. If the <alternative question> is an <expression>, the *Range-condition* of the <answer>s in the <decision body> must be of the same sort as of the <alternative question>.

There is syntactic ambiguity between <informal text> and <character string> in <alternative question> and <answer>s in the <decision body>. If the <alternative question> and all <answer>s are <character string>, all of these are interpreted as <informal text>. If the <alternative question> or any <answer> is a <character string> and this does not match the context of the transition option, the <character string> denotes <informal text>.

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

Concrete graphical grammar

```
<transition option area> ::=
                        <transition option symbol> contains <alternative question>
                        is followed by <graphical decision body>

<transition option symbol> ::=
```



The <flow line symbol>s in <graphical decision body> are 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>s in the <graphical decision body> must be mutually exclusive.

Semantics

Constructs in a <graphical answer part> are selected if the <answer> contains the result of the <alternative question>. If none of the <answer>s contains the result of the <alternative question>, then the constructs in the <graphical else part> are selected.

If no <graphical else part> 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 11.13.5.

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

ANNEX A

Index of non-terminals

The following non-terminals are intentionally defined and not used: <macro call>, <page>, <comment area>, <text extension area>, <sdl specification>.

ANNEX B

Reserved for future use

ANNEX C

Reserved for future use

ANNEX D

SDL Predefined data

D.1 Introduction

Predefined data 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. Even though the definition of an abstract data type gives one possible way of implementing that data type, an implementation is not mandated to choose that way of implementing the abstract data type, as long as the same abstract behaviour is preserved.

The predefined data types, including the Boolean sort which defines properties for two literals true and false, are defined in this annex. The two Boolean *terms* true and false must not be (directly or indirectly) defined to be equivalent. Every Boolean constant expression that 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.

Predefined data is defined in an implicitly used package Predefined (see 7.2). This package is defined in this annex.

D.2 Notation

For this purpose, this annex extends the concrete syntax of SDL by means of describing the abstract properties of the operations added by a data type definition. However, this additional syntax is used for explanation only and does not extend the syntax defined in the main text. A specification using the syntax defined in this annex is therefore not valid SDL.

The abstract properties described here do not specify a specific representation of the predefined data. Instead, an interpretation must conform to these properties. When an <expression> is interpreted, the evaluation of the expression produces a value (e.g. as the result of an <operation application>). Two expressions E1 and E2 are equivalent, if:

a) there is an <equation> E1 = = E2, or

b) one of the equations derived from the given set of <quantified equations>s is E1 = = E2; or

c) i) E1 is equivalent to EA; and

   ii) E2 is equivalent to EB; and

   iii) there is an equation or an equation derived from the given set quantified equations such that EA = = EB; or

d) by substituting a sub-term of E1 by a term of the same class as the sub-term producing a term E1A it is possible to show that E1A is in the same class as E2.

Otherwise, the two expression are not equivalent.

Two expressions that are equivalent represent the same value.

Interpretation of expressions conforms to these properties if two equivalent expressions represent the same value, and two non-equivalent expressions represent different values.

D.2.1 Axioms

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.

An operation defined by <axiomatic operation definitions> is treated as a complete definition with respect to specialization. That is, when a data type defined by the package Predefined is specialized and an operation is redefined in the specialized type, all axioms mentioning the name of the operation are replaced by the corresponding definition in the specialized type.

Concrete textual grammar

```
<axiomatic operation definitions> ::=
                              axioms <axioms>

<axioms> ::=
                              <equation> { <end> <equation>}* [<end> ]

<equation> ::=
                              <unquantified equation>
                        |     <quantified equations>
                        |     <conditional equation>
                        |     <literal equation>
                        |     <noequality>

<unquantified equation> ::=
                              <term> == <term>
                        |     <Boolean axiom>

<term> ::=
                              <constant expression>
                        |     <error term>

<quantified equations> ::=
                              <quantification> (<axioms> )

<quantification> ::=
                              for all <value name> { , <value name> }* in <sort>
```

This annex changes <operations> (see 12.1.1) as described below.

```
<operations> ::=
                              <operation signatures>
                              { <operation definitions> | <axiomatic operation definitions> }
```

<axiomatic operation definitions> can only be used to describe the behaviour of operators.

An <identifier> which is an unqualified name appearing in a <term>can be:

a) an <operation identifier> (see 12.2.7);

b) a <literal identifier> (see 12.2.2);

c) 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

d) a <value identifier> which has an implied quantified equation 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>, within the <conditional equation>), imply one <quantification>.

Within one <unquantified equation> (or in case the <unquantified equation> is contained in a <conditional equation>, within the <conditional equation>), there must be exactly one sort for each implicitly quantified value identifier which is consistent with all its uses.

Semantics

A ground term is a term that 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.

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.

In general, there is no need or reason to distinguish between a ground term and the result of the ground term. For example, the ground term for the unity Integer element can be written "1". Usually there are several ground terms which denote the same data item, e.g. the Integer ground terms "0+1", "3–2" and "(7+5)/12", and it is usual to consider a simple form of the ground term (in this case "1") as denoting the data item.

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, 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 in X (for all z in X ... )
```

introduces only one value identifier named z of sort X.

In the concrete syntax of axioms, 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>. 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 that has implied quantification allow different sorts, the identifier is bound to a sort that 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.

D.2.2 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.

Concrete textual grammar

```
<conditional equation> ::=
                        <restriction> { , <restriction> }* ==><restricted equation>

<restricted equation> ::=
                        <unquantified equation>

<restriction> ::=
                        <unquantified equation>
```

Semantics

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

The semantics of a set of equations for a data type that includes conditional equations are derived as follows:

a) Quantification is removed by generating every possible ground term equation that 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 that are not restricted equations. If there exists a provable conditional equation, 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.

D.2.3 Equality

Concrete textual grammar

```
<noequality> ::=
                        noequality
```

Model

Any <data type definition> introducing some sort named S has the following implied <operation signature> in its <operator list>, unless <noequality> is present in the <axioms>:

```
equal ( S, S ) -> Boolean;
```

where Boolean is the predefined Boolean sort.

Any <data type definition> introducing a sort named S such that it contains only <axiomatic operation definitions> in <operator list> has an implied equation set:

```
for all a,b,c in S (
   equal(a, a) == true;
   equal(a, b) == equal(b, a);
   equal(a, b) and equal(b, c)  ==> equal(a, c) == true;
   equal(a, b) == true          ==> a == b;)
```

and an implied <literal equation>:

```
   for all L1,L2 in S literals (
      spelling(L1) /= spelling(L2) ==> L1 = L2 == false;)
```

D.2.4 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.

Model

An axiom of the form:

```
        <Boolean term>;
```

is derived syntax for the concrete syntax equation:

```
        <Boolean term> == << package Predefined/type Boolean >> true;
```

D.2.5 Conditional 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 <conditional expression> by each ground term of the appropriate sort. In this set of equations the <conditional expression> 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 that contains:

a) for every *equation* in the expanded ground set for which the <conditional expression> is equivalent to true, that *equation* from the expanded ground set with the <conditional expression> replaced by the (ground) <consequence expression>; and

b) for every *equation* in the expanded ground set for which the <conditional expression> is equivalent to false, that *equation* from the expanded ground set with the <conditional expression> replaced by the (ground) <alternative expression>.

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;
```

D.2.6 Error term

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.

Concrete textual grammar

```
<error term> ::=
                         raise <exception name>
```

An <error term> must not be used as part of a <restriction>.

It must not be possible to derive from *equations* that a <literal identifier> is equal to <error term>.

Semantics

A term may be an <error term> so that it is possible to specify the circumstances under which an operator produces an error. If these circumstances arise during interpretation, then the exception with <exception name> is raised.

D.2.7 Unordered literals

Concrete textual grammar

```
<unordered> ::=
                    unordered
```

This annex changes the concrete syntax for the literal list type constructor (see 12.1.7.1) as follows.

```
<literal list> ::=
                    [<protected>] literals [<unordered>]
                         <literal signature> { , <literal signature> }* <end>
```

Model

If <unordered> is used, the *Model* in 12.1.7.1 is not applied. Consequentially, the ordering operations "<", ">","<=",">=", first, last, pred, succ, and num are not implicitly defined for this data type.

D.2.8 Literal equations

Concrete textual grammar

```
<literal equation> ::=
                    <literal quantification>
                         ( <equation> { <end> <equation> }* [<end>] )

<literal quantification> ::=
                    for all <value name> { , <value name> }* in <sort> literals
               |    for all <value name> { , <value name> }* in { <sort> | <value identifier> } nameclass
```

This annex changes the concrete syntax for <spelling term> (see 12.1.9.2) as follows.

```
<spelling term> ::=
                    spelling ( { <operation name> | <value identifier> } )
```

The <value identifier> in a <spelling term> must be a <value identifier> defined by a <literal quantification>.

Semantics

Literal mapping is a shorthand for defining a large (possibly infinite) number of axioms ranging over all the literals of a sort or all the names in a name class. The literal mapping allows the literals for a sort to be mapped onto the values of the sort.

<spelling term> is used in literal quantifications 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 equations.

Model

A <literal equation> is shorthand for a set of <axioms>. In each of <equation>s contained in a <literal equation>, 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 identifier> of the <sort> of the <literal quantification> (if **literals** was used) or by the same <literal identifier> of the nameclass referred to (if **nameclass** was used). The derived set of <axioms> contains all possible <equation>s that can be derived in this way.

The derived <axioms> for <literal equation>s are added to <axioms> (if any) defined after the keyword **axioms**.

If a <literal quantification> contains one or more <spelling term>s, then there is replacement of the <spelling term>s with Charstring literals (see D.3). The Charstring is used to replace the <value identifier> after the <literal equation> containing the <spelling term> and is expanded as defined in 12.1.9.2, using <value identifier> in place of <operation name>.

NOTE – Literal equations do not affect nullary operators defined in <operation signature>s.

D.3 Package Predefined

In the following definitions, all references to names defined in the package Predefined are considered to be treated as prefixed by the qualification <<**package** Predefined>>. To increase readability, this qualification is omitted.

```
/* */
package Predefined
/*
```

D.3.1 Boolean sort

D.3.1.1 Definition

```
*/
value type Boolean
  literals true, false;
  operators
     "not" ( this Boolean )                -> this Boolean;
     "and" ( this Boolean, this Boolean )  -> this Boolean;
     "or"  ( this Boolean, this Boolean )  -> this Boolean;
     "xor" ( this Boolean, this Boolean )  -> this Boolean;
     "=>"  ( this Boolean, this Boolean )  -> this Boolean;
axioms
     not( true  )     == false;
     not( false )     == true ;
/* */
     true  and true   == true ;
     true  and false  == false;
     false and true   == false;
     false and false  == false;
/* */
     true  or true    == true ;
     true  or false   == true ;
     false or true    == true ;
     false or false   == false;
/* */
     true  xor true   == false;
     true  xor false  == true ;
     false xor true   == true ;
     false xor false  == false;
/* */
     true  => true    == true ;
     true  => false   == false;
     false => true    == true ;
     false => false   == true ;
endvalue type Boolean;
/*
```

D.3.1.2 Usage

The Boolean sort is used to represent true and false values. Often it is used as the result of a comparison.

The Boolean sort is used widely throughout SDL.

D.3.2 Character sort

D.3.2.1 Definition

```
*/
value type Character
  literals
      NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL,
      BS,  HT,  LF,  VT,  FF,  CR,  SO,  SI,
      DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB,
      CAN, EM,  SUB, ESC, IS4, IS3, IS2, IS1,
      ' ', '!', '"', '#', '$', '%', '&', '''',
      '(', ')', '*', '+', ',', '-', '.', '/',
      '0', '1', '2', '3', '4', '5', '6', '7',
      '8', '9', ':', ';', '<', '=', '>', '?',
      '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
      'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
      'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
      'X', 'Y', 'Z', '[', '\', ']', '^', '_',
      '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
      'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
      'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
      'x', 'y', 'z', '{', '|', '}', '~', DEL;
/* '''' is an apostrophe, ' ' is a space, '˜' is a tilde */
/* */
  operators
      chr  ( Integer ) -> this Character;
/*   "<", "<=", ">", ">=", and "num" are implicitly defined (see 12.1.7.1). */
axioms
      for all a,b in Character (
      for all i in Integer (
/* definition of Chr */
      chr(num(a)) == a;
      chr(i+128)  == chr(i);
      ));
endvalue type Character;
/*
```

D.3.2.2 Usage

The Character sort is used to represent characters of the International Reference Alphabet (Recommendation T.50).

D.3.3 String sort

D.3.3.1 Definition

```
*/
value type String < type Itemsort >
/* Strings are "indexed" from one */
  operators
      emptystring                                    -> this String;
      mkstring   ( Itemsort                      )  -> this String;
      make       ( Itemsort                      )  -> this String;
      length     ( this String                     ) -> Integer;
      first      ( this String                     ) -> Itemsort;
      last       ( this String                     ) -> Itemsort;
      "//"       ( this String, this String        ) -> this String;
      extract    ( this String, Integer            ) -> Itemsort raise InvalidIndex;
      modify     ( this String, Integer, Itemsort  ) -> this String;
      substring  ( this String, Integer, Integer )  -> this String raise InvalidIndex;
      /* substring (s,i,j) gives a string of length j starting from the ith element */
      remove     ( this String, Integer, Integer )  -> this String;
      /* remove (s,i,j) gives a string with a substring of length j starting from
         the ith element removed */
axioms
      for all e in Itemsort ( /*e – element of Itemsort*/
      for all s,s1,s2,s3 in String (
      for all i,j in Integer (
 /* constructors are emptystring, mkstring, and "//" */
 /* equalities between constructor terms */
      s // emptystring== s;
      emptystring // s== s;
      (s1 // s2) // s3== s1 // (s2 // s3);
 /* */
 /* definition of length by applying it to all constructors */
      <<type String>>length(emptystring)== 0;
      <<type String>>length(mkstring(e))== 1;
      <<type String>>length(s1 // s2)    == length(s1) + length(s2);
      make(s)                             == mkstring(s);
 /* */
 /* definition of extract by applying it to all constructors,
         error cases handled separately */
      extract(mkstring(e),1)                  == e;
      i <= length(s1) ==> extract(s1 // s2,i)== extract(s1,i);
      i > length(s1) ==> extract(s1 // s2,i)  == extract(s2,i-length(s1));
      i<=0 or i>length(s) ==> extract(s,i)   == raise InvalidIndex;
 /* */
 /* definition of first and last by other operations */
      first(s)  == extract(s,1);
      last(s)   == extract(s,length(s));
 /* */
 /* definition of substring(s,i,j) by induction on j,
         error cases handled separately */
      i>0 and i-1<=length(s)           ==>
                   substring(s,i,0)  == emptystring;
 /* */
      i>0 and j>0 and i+j-1<=length(s)  ==>
                      substring(s,i,j)  == substring(s,i,j-1) // mkstring(extract(s,i+j-1));
 /* */
      i<=0 or j<0 or i+j-1>length(s)    ==>
                      substring(s,i,j)  == raise InvalidIndex;
 /* */
 /* definition of modify by other operations */
      modify(s,i,e)== substring(s,1,i-1) // mkstring(e) // substring(s,i+1,length(s)-i);
 /* definition of remove */
      remove(s,i,j)   == substring(s,1,i-1) // substring(s,i+j,length(s)-i-j+1);
      )));
 endvalue type String;
 /*
```

D.3.3.2 Usage

The make, extract, and modify operators will typically be used with the shorthands defined in 12.2.4 and 12.3.3.1 for accessing the values of strings and assigning values to strings.

D.3.4 Charstring sort

D.3.4.1 Definition

```
*/
value type Charstring
     inherits String < Character > ( '' = emptystring )
     adding
        operators ocs in nameclass
             '''' ( (' ':'&') or '''''' or ('(': '~') )+ '''' -> this Charstring;
/* character strings of any length of any characters from a space ' ' to a tilde '~' */
axioms
     for all c in Character nameclass (
     for all cs, cs1, cs2 in ocs nameclass (
     spelling(cs) == spelling(c)                     ==> cs == mkstring(c);
/* string 'A' is formed from character 'A' etc. */
     spelling(cs) == spelling(cs1) // spelling(cs2),
     length(spelling(cs2)) == 1                     ==> cs  == cs1 // cs2;
     ));
endvalue type Charstring;
/*
```

D.3.4.2 Usage

```
/*
```

The Charstring sort defines strings of characters.

A Charstring literal can contain printing characters and spaces.

A non printing character can be used as a string by using mkstring, for example mkstring(DEL).

Example:

```
synonym newline_prompt Charstring = mkstring(CR) // mkstring(LF) // '$>';
```

D.3.5 Integer sort

D.3.5.1 Definition

```
*/
value type Integer
  literals unordered nameclass (('0':'9')*) ('0':'9'));
  operators
     "-"  ( this Integer                ) -> this Integer;
     "+"  ( this Integer, this Integer ) -> this Integer;
     "-"  ( this Integer, this Integer ) -> this Integer;
     "*"  ( this Integer, this Integer ) -> this Integer;
     "/"  ( this Integer, this Integer ) -> this Integer raise DivisionByZero;
     "mod"( this Integer, this Integer ) -> this Integer raise DivisionByZero;
     "rem"( this Integer, this Integer ) -> this Integer;
     "<"  ( this Integer, this Integer ) -> Boolean;
     ">"  ( this Integer, this Integer ) -> Boolean;
     "<=" ( this Integer, this Integer ) -> Boolean;
     ">=" ( this Integer, this Integer ) -> Boolean;
     power( this Integer, this Integer ) -> this Integer;
     bs in nameclass '''' ( (('0' or '1')*'''B') or ((('0':'9') or ('A':'F'))*'''H') )
              -> this Integer;
axioms  noequality
     for all a,b,c in Integer (
/* constructors are 0, 1, +, and unary - */
/* equalities between constructor terms */
     (a + b) + c          == a + (b + c);
     a + b                == b + a;
     0 + a                == a;
     a + (- a)            == 0;
     (- a) + (- b)      == - (a + b);
     <<type Integer>> - 0 == 0;
     - (- a)              == a;
/* */
/* definition of binary "-" by other operations */
     a - b                == a + (- b);
/* */
/* definition of "*" by applying it to all constructors */
     0 * a                == 0;
     1 * a                == a;
     (- a) * b            == - (a * b);
     (a + b) * c          == a * c + b * c;
/* */
/* definition of "<" by applying it to all constructors */
     a < b                                  == 0 < (b - a);
     <<type Integer>> 0 < 0                 == false;
     <<type Integer>> 0 < 1                 == true ;
     0 < a           == true ==> 0 < (- a)   == false;
     0 < a and 0 < b == true ==> 0 < (a + b) == true ;
/* */
/* definition of ">", "equal", "<=", and ">=" by other operations */
     a > b                == b < a;
     equal(a, b)          == not(a < b or a > b);
     a <= b               == a < b or a = b;
     a >= b               == a > b or a = b;
/* */
/* definition of "/" by other operations */
     a / 0                                        == raise DivisionByZero;
     a >= 0 and b > a          == true ==> a / b == 0;
     a >= 0 and b <= a and b > 0  == true ==> a / b == 1 + (a-b) / b;
     a >= 0 and b < 0          == true ==> a / b == - (a / (- b));
     a < 0 and b < 0             == true ==> a / b == (- a) / (- b);
     a < 0 and b > 0             == true ==> a / b == - ((- a) / b);
/* */
/* definition of "rem" by other operations */
     a rem b == a - b * (a/b);
/* */
/* definition of "mod" by other operations */
     a >= 0 and b > 0               ==> a mod b  == a rem b;
     b < 0                            ==> a mod b  == a mod (- b);
     a < 0 and b > 0 and a rem b = 0 ==> a mod b  == 0;
     a < 0 and b > 0 and a rem b < 0 ==> a mod b  == b + a rem b;
     a mod 0 == raise DivisionByZero;
/* */
/* definition of power by other operations */
     power(a, 0)           == 1;
     b > 0 ==> power(a, b)  == a * power(a, b-1);
     b < 0 ==> power(a, b)  == power(a, b+1) / a; );
/* */
/* definition of literals */
     <<type Integer>> 2 == 1 + 1;
     <<type Integer>> 3 == 2 + 1;
     <<type Integer>> 4 == 3 + 1;
     <<type Integer>> 5 == 4 + 1;
     <<type Integer>> 6 == 5 + 1;
     <<type Integer>> 7 == 6 + 1;
     <<type Integer>> 8 == 7 + 1;
     <<type Integer>> 9 == 8 + 1;
/* */
/* literals other than 0 to 9 */
     for all a,b,c in Integer nameclass (
     spelling(a) == spelling(b) // spelling(c),
     length(spelling(c)) == 1                ==> a == b * (9 + 1) + c;
     );
/* */
/* hex and binary representation of Integer */
     for all b in Bitstring nameclass (
     for all i in bs nameclass (
     spelling(i) == spelling(b)             ==> i == <<type Bitstring>>num(b);
     ));
endvalue type Integer;
/*
```

D.3.5.2 Usage

The Integer sort is used for mathematical integers with decimal, hex, or binary notation.

D.3.6 Natural syntype

D.3.6.1 Definition

```
*/
syntype Natural = Integer constants >= 0 endsyntype Natural;
/*
```

D.3.6.2 Usage

The natural syntype is used when positive integers only are required. All operators will be the integer operators but when a value is used as a parameter or assigned the value is checked. A negative value will be an error.

D.3.7 Real sort

D.3.7.1 Definition

```
*/
value type Real
  literals unordered nameclass
        ( ('0':'9')* ('0':'9') ) or ( ('0':'9')* '.'('0':'9')+ );
  operators
      "-"  ( this Real            ) -> this Real;
      "+"  ( this Real, this Real ) -> this Real;
      "-"  ( this Real, this Real ) -> this Real;
      "*"  ( this Real, this Real ) -> this Real;
      "/"  ( this Real, this Real ) -> this Real raise DivisionByZero;
      "<"  ( this Real, this Real ) -> Boolean;
      ">"  ( this Real, this Real ) -> Boolean;
      "<=" ( this Real, this Real ) -> Boolean;
      ">=" ( this Real, this Real ) -> Boolean;
      float( Integer              ) -> this Real;
      fix  ( this Real            ) -> Integer;
axioms noequality
      for all r,s in Real (
      for all a,b,c,d in Integer (
/* constructors are float and "/" */
/* equalities between constructor terms allow to reach always a form
        float(a) / float(b) where b > 0 */
      r / float(0)                                 == raise DivisionByZero;
      r / float(1)                                 == r;
      c /= 0 ==> float(a) / float(b)               == float(a*c) / float(b*c);
      b /= 0 and d /= 0 ==>
      (float(a) / float(b)) / (float(c) / float(d))  == float(a*d) / float(b*c);
/* */
/* definition of unary "-" by applying it to all constructors */
      - (float(a) / float(b))                      == float(- a) / float(b);
/* */
/* definition of "+" by applying it to all constructors */
      (float(a) / float(b)) + (float(c) / float(d))  ==float(a*d + c*b) / float(b*d);
/* */
/* definition of binary "-" by other operations */
      r - s                                        == r + (- s);
/* */
/* definition of "*" by applying it to all constructors */
      (float(a) / float(b)) * (float(c) / float(d))  == float(a*c) / float(b*d);
/* */
/* definition of "<" by applying it to all constructors */
      b > 0 and d > 0 ==>
      (float(a) / float(b)) < (float(c) / float(d))  == a * d < c * b;
/* */
/* definition of ">", "equal", "<=", and ">=" by other operations */
      r > s       == s < r;
      equal(r, s) == not(r < s or r > s);
      r <= s      == r < s or r = s;
      r >= s      == r > s or r = s;
/* */
/* definition of fix by applying it to all constructors */
      a >= b and b > 0==> fix(float(a) / float(b)) == fix(float(a-b) / float(b)) + 1;
      b > a  and a >= 0  ==> fix(float(a) / float(b))== 0;
      a < 0  and b > 0==> fix(float(a) / float(b)) == - fix(float(-a)/float(b)) - 1;));
/* */
      for all r,s in Real nameclass (
      for all i,j in Integer nameclass (
         spelling(r) == spelling(i)          ==> r == float(i);
/* */
         spelling(r) == spelling(i)          ==> i == fix(r);
/* */
         spelling(r) == spelling(i) // spelling(s),
        spelling(s) == '.' // spelling(j) ==> r == float(i) + s;
/* */
         spelling(r) == '.' // spelling(i),
        length(spelling(i)) == 1            ==> r == float(i) / 10;
/* */
        spelling(r) == '.' // spelling(i) // spelling(j),
        length(spelling(i)) == 1,
        spelling(s) == '.' // spelling(j) ==> r == (float(i) + s) / 10;
     ));
endvalue type Real;
/*
```

D.3.7.2 Usage

The real sort is used to represent real numbers.

The real sort can represent all numbers which can be represented as one integer divided by another.

Numbers which cannot be represented in this way (irrational numbers – for example the square root of 2) are not part of the real sort. However for practical engineering a sufficiently accurate approximation can usually be used.

D.3.8 Array sort

D.3.8.1 Definition

```
*/
value type Array < type Index; type Itemsort >
  operators
     make                                -> this Array ;
     make   ( Itemsort                 ) -> this Array ;
     modify ( this Array,Index,Itemsort ) -> this Array ;
     extract( this Array,Index         ) -> Itemsort raise InvalidIndex;
axioms
     for all item, itemi, itemj in Itemsort (
     for all i, j in Index (
     for all a, s in Array (
        <<type Array>>extract(make,i)                    == raise InvalidIndex;
        <<type Array>>extract(make(item),i)              == item ;
        i = j ==> modify(modify(s,i,itemi),j,item)       == modify(s,i,item);
        i = j ==> extract(modify(a,i,item),j)            == item ;
        i = j == false ==> extract(modify(a,i,item),j)   == extract(a,j);
        i = j == false ==> modify(modify(s,i,itemi),j,itemj)==
                                                    modify(modify(s,j,itemj),i,itemi);
/*equality*/
        <<type Array>>make(itemi) = make(itemj)          == itemi = itemj;
        a=s == true, i=j == true, itemi = itemj ==>
                 modify(a,i,itemi) = modify(s,j,itemj)   == true;
/* */
        extract(a,i) = extract(s,i) == false ==>  a = s  == false;)));
endvalue type Array;
/*
```

D.3.8.2 Usage

An array can be used to define one sort which is indexed by another. For example:

```
        value type indexbychar inherits Array< Character, Integer >
        endvalue type indexbychar;
```

defines an array containing integers and indexed by characters.

Arrays are usually used in combination with the shorthand forms of make, modify, and extract defined in 12.2.4 and 12.3.3.1. For example:

```
        dcl charvalue indexbychar;
        task charvalue := (. 12 .);
        task charvalue('A') := charvalue('B')-1;
```

D.3.9 Vector

D.3.9.1 Definition

```
*/
value type Vector < type Itemsort; synonym MaxIndex >
     inherits Array< Indexsort, Itemsort >;
syntype Indexsort = Integer constants 1:MaxIndex endsyntype;
endvalue type Vector;
/*
```

D.3.10 Powerset sort

D.3.10.1 Definition

```
*/
value type Powerset < type Itemsort >
  operators
     empty                                    -> this Powerset;
     "in"   ( Itemsort, this Powerset       ) -> Boolean;       /* is member of          */
     incl   ( Itemsort, this Powerset       ) -> this Powerset; /* include item in set   */
     del    ( Itemsort, this Powerset       ) -> this Powerset; /* delete item from set  */
     "<"    ( this Powerset, this Powerset  ) -> Boolean;       /* is proper subset of   */
     ">"    ( this Powerset, this Powerset  ) -> Boolean;       /* is proper superset of */
     "<="   ( this Powerset, this Powerset  ) -> Boolean;       /* is subset of          */
     ">="   ( this Powerset, this Powerset  ) -> Boolean;       /* is superset of        */
     "and"  ( this Powerset, this Powerset  ) -> this Powerset; /* intersection of sets  */
     "or"   ( this Powerset, this Powerset  ) -> this Powerset; /* union of sets         */
     length ( this Powerset                 ) -> Integer;
     take   ( this Powerset                 ) -> Itemsort raise Empty;
axioms
     for all i,j in Itemsort (
     for all p,ps,a,b,c in Powerset (
/* constructors are empty and incl */
/* equalities between constructor terms */
     incl(i,incl(j,p))                   == incl(j,incl(i,p));
     i = j ==> incl(i,incl(j,p))         == incl(i,p);
/* definition of "in" by applying it to all constructors */
     i in <<type Powerset>>empty         == false;
     i in incl(j,ps)                     == i=j or i in ps;
/* definition of del by applying it to all constructors */
     <<type Powerset>>del(i,empty)       == empty;
     i =  j ==> del(i,incl(j,ps))      == del(i,ps);
     i /= j ==> del(i,incl(j,ps))      == incl(j,del(i,ps));
/* definition of "<" by applying it to all constructors */
     a < <<type Powerset>>empty        == false;
     <<type Powerset>>empty < incl(i,b)== true;
     incl(i,a) < b                     == i in b and del(i,a) < del(i,b);
/* definition of ">" by other operations */
     a > b == b < a;
/* definition of "=" by applying it to all constructors */
     empty = incl(i,ps)                == false;
     incl(i,a) = b     == i in b and del(i,a) = del(i,b);
/* definition of "<=" and ">=" by other operations */
     a <= b                              == a < b or a = b;
     a >= b                              == a > b or a = b;
/* definition of "and" by applying it to all constructors */
     empty and b                         == empty;
     i in b     ==> incl(i,a) and b    == incl(i,a and b);
     not(i in b) ==> incl(i,a) and b   == a and b;
/* definition of "or" by applying it to all constructors */
     empty or b                          == b;
     incl(i,a) or b                      == incl(i,a or b);
/* definition of length */
     length(<<type Powerset>>empty)      == 0;
     i in ps ==> length(ps)              == 1 + length(del(i, ps));
/* definition of take */
     take(empty)                         == raise Empty;
     i in ps ==> take(ps)                == i;
     ));
endvalue type Powerset;
/*
```

D.3.10.2 Usage

Powersets are used to represent mathematical sets. For example:

```
        value type Boolset inherits Powerset< Boolean > endvalue type Boolset;
```

can be used for a variable which can be empty or contain (true), (false) or (true, false).

D.3.11 Duration sort

D.3.11.1 Definition

```
*/
value type Duration
  literals unordered nameclass ('0':'9')+ or (('0':'9')* '.' ('0':'9')+);
  operators
     protected duration ( Real             ) -> this Duration;
     "+"  ( this Duration, this Duration ) -> this Duration;
     "-"  ( this Duration                ) -> this Duration;
     "-"  ( this Duration, this Duration ) -> this Duration;
     ">"  ( this Duration, this Duration ) -> Boolean;
     "<"  ( this Duration, this Duration ) -> Boolean;
     ">=" ( this Duration, this Duration ) -> Boolean;
     "<=" ( this Duration, this Duration ) -> Boolean;
     "*"  ( this Duration, Real          ) -> this Duration;
     "*"  ( Real, this Duration          ) -> this Duration;
     "/"  ( this Duration, Real          ) -> Duration;
axioms noequality
/* constructor is duration(Real)*/
     for all a, b in Real nameclass (
     for all d, e in Duration nameclass (
/* definition of "+" by applying it to all constructors */
     duration(a) + duration(b)    == duration(a + b);
/* */
/* definition of unary "-" by applying it to all constructors */
     - duration(a)             == duration(-a);
/* */
/* definition of binary "-" by other operations */
     d - e                        == d + (-e);
/* */
/* definition of "equal", ">", "<", ">=", and "<=" by applying it to all constructors */
     equal(duration(a), duration(b)) == a = b;
     duration(a) > duration(b)    == a > b;
     duration(a) < duration(b)    == a < b;
     duration(a) >= duration(b) == a >= b;
     duration(a) <= duration(b) == a <= b;
/* */
/* definition of "*" by applying it to all constructors */
     duration(a) * b              == duration(a * b);
     a * d                        == d * a;
/* */
/* definition of "/" by applying it to all constructors */
     duration(a) / b            == duration(a / b);
/* */
     spelling(d) == spelling(a) ==>
                             d  == duration(a);
     ));
endvalue type Duration;
/*
```

D.3.11.2 Usage

The duration sort is used for the value to be added to the current time to set timers. The literals of the sort duration are the same as the literals for the real sort. The meaning of one unit of duration will depend on the system being defined.

Duration values can be multiplied and divided by real values.

D.3.12 Time sort

D.3.12.1 Definition

```
*/
value type Time
  literals unordered nameclass ('0':'9')+ or (('0':'9')* '.' ('0':'9')+);
  operators
     protected time  ( Duration   ) -> this Time;
     "<"  ( this Time, this Time  ) -> Boolean;
     "<=" ( this Time, this Time  ) -> Boolean;
     ">"  ( this Time, this Time  ) -> Boolean;
     ">=" ( this Time, this Time  ) -> Boolean;
     "+"  ( this Time, Duration) -> this Time;
     "+"  ( Duration, this Time) -> this Time;
     "-"  ( this Time, Duration) -> this Time;
     "-"  ( this Time, this Time  ) -> Duration;
axioms noequality
/* constructor is time */
     for all t, u in Time nameclass (
     for all a, b in Duration nameclass (
/* definition of ">", "equal" by applying it to all constructors */
     time(a) > time(b)          == a > b;
     equal(time(a), time(b))    == a = b;
/* */
/* definition of "<", "<=", ">=" by other operations */
     t < u                      == u > t;
     t <= u                     == (t < u) or (t = u);
     t >= u                     == (t > u) or (t = u);
/* */
/* definition of "+" by applying it to all constructors */
     time(a) + b                == time(a + b);
     a + t                      == t + a;
/* */
/* definition of "-" : Time, Duration by other operations */
     t - b                      == t + (-b);
/* */
/* definition of "-" : Time, Time by applying it to all constructors */
     time(a) - time(b)          == a - b;
/* */
     spelling(a) == spelling(t) ==>
                            a  == time(t);
     ));
endvalue type Time;
/*
```

D.3.12.2 Usage

The **now** expression returns a value of the time sort. A time value may have a duration added or subtracted from it to give another time. A time value subtracted from another time value gives a duration. Time values are used to set the expiry time of timers.

The origin of time is system dependent. A unit of time is the amount of time represented by adding one duration unit to a time.

D.3.13 Bag sort

D.3.13.1 Definition

```
*/
value type Bag < type Itemsort >
  operators
     empty                          -> this Bag;
     "in"   ( Itemsort, this Bag  ) -> Boolean;/* is member of  */
     incl   ( Itemsort, this Bag  ) -> this Bag;  /* include item in set     */
     del    ( Itemsort, this Bag  ) -> this Bag;  /* delete item from set    */
     "<"    ( this Bag, this Bag  ) -> Boolean;/* is proper subbag of       */
     ">"    ( this Bag, this Bag  ) -> Boolean;/* is proper superbag of     */
     "<="   ( this Bag, this Bag  ) -> Boolean;/* is subbag of  */
     ">="   ( this Bag, this Bag  ) -> Boolean;/* is superbag of            */
     "and"  ( this Bag, this Bag  ) -> this Bag;  /* intersection of bags    */
     "or"   ( this Bag, this Bag  ) -> this Bag;  /* union of bags           */
     length ( this Bag            ) -> Integer;
     take   ( this Bag            ) -> Itemsort raise Empty;
axioms
     for all i,j in Itemsort (
     for all p,ps,a,b,c in Bag (
/* constructors are empty and incl */
/* equalities between constructor terms */
     incl(i,incl(j,p))               == incl(j,incl(i,p));
/* definition of "in" by applying it to all constructors */
     i in <<type Bag>>empty          == false;
     i in incl(j,ps)                 == i=j or i in ps;
/* definition of del by applying it to all constructors */
     <<type Bag>>del(i,empty)        == empty;
     i =  j ==> del(i,incl(j,ps))  == ps;
     i /= j ==> del(i,incl(j,ps))  == incl(j,del(i,ps));
/* definition of "<" by applying it to all constructors */
     a < <<type Bag>>empty           == false;
     <<type Bag>>empty < incl(i,b)   == true;
     incl(i,a) < b                 == i in b and del(i,a) < del(i,b);
/* definition of ">" by other operations */
     a > b == b < a;
/* definition of "=" by applying it to all constructors */
     empty = incl(i,ps)            == false;
     incl(i,a) = b                 == i in b and del(i,a) = del(i,b);
/* definition of "<=" and ">=" by other operations */
     a <= b                          == a < b or a = b;
     a >= b                          == a > b or a = b;
/* definition of "and" by applying it to all constructors */
     empty and b                     == empty;
     i in b     ==> incl(i,a) and b == incl(i,a and b);
     not(i in b) ==> incl(i,a) and b == a and b;
/* definition of "or" by applying it to all constructors */
     empty or b                      == b;
     incl(i,a) or b                  == incl(i,a or b);
/* definition of length */
     length(<<type Bag>>empty)       == 0;
     i in ps    ==> length(ps)       == 1 + length(del(i, ps));
/* definition of take */
     take(empty)                     == raise Empty;
     i in ps    ==> take(ps)         == i; ));
endvalue type Bag;
/*
```

D.3.13.2 Usage

Bags are used to represent multi-sets. For example:

```
     value type Boolset Bag< Boolean > endvalue type Boolset;
```

can be used for a variable which can be empty or contain (true), (false), (true, false) (true, true), (false, false),... .

Bags are used to represent the SET OF construction of ASN.1.

D.3.14 ASN.1 Bit and Bitstring sorts

D.3.14.1 Definition

```
*/
value type Bit
     inherits Boolean ( 0 = false, 1 = true );
     adding
  operators
     num ( this Bit  ) -> Integer;
     bit ( Integer ) -> this Bit raise OutOfRange;
```

```
axioms
     <<type Bit>>num (0)           == 0;
     <<type Bit>>num (1)           == 1;
     <<type Bit>>bit (0)           == 0;
     <<type Bit>>bit (1)           == 1;
     for all i in Integer (
        i > 1 or i < 0 ==> bit (i)== raise OutOfRange;
        )
endvalue type Bit;
/* */
value type Bitstring
  operators
     bs in nameclass
     '''' ( (('0' or '1')*'''B') or ((('0':'9') or ('A':'F'))*'''H') )-> this Bitstring;
/*The following operators that are the same as String except Bitstring
     is indexed from zero*/
     mkstring  (Bit                               ) -> this Bitstring;
     make      (Bit                               ) -> this Bitstring;
     length    ( this Bitstring                   ) -> Integer;
     first     ( this Bitstring                   ) -> Bit;
     last      ( this Bitstring                   ) -> Bit;
     "//"      ( this Bitstring, this Bitstring    ) -> this Bitstring;
     extract   ( this Bitstring, Integer          ) -> Bit raise InvalidIndex;
     modify    ( this Bitstring, Integer, Bit      ) -> this Bitstring;
     substring ( this Bitstring, Integer, Integer) -> this Bitstring raise InvalidIndex;
     /* substring (s,i,j) gives a string of length j starting from the ith element */
     remove    ( this Bitstring, Integer, Integer) -> this Bitstring;
     /* remove (s,i,j) gives a string with a substring of length j starting from
        the ith element removed */
/*The following operators are specific to Bitstrings*/
     "not" ( this Bitstring                ) ->  this Bitstring;
     "and" ( this Bitstring, this Bitstring  ) ->  this Bitstring;
     "or"  ( this Bitstring, this Bitstring  ) ->  this Bitstring;
     "xor" ( this Bitstring, this Bitstring  ) ->  this Bitstring;
     "=>"  ( this Bitstring, this Bitstring  ) ->  this Bitstring
     num   ( this Bitstring                ) ->  Integer;
     bitstring ( Integer                   ) ->  this Bitstring raise OutOfRange;
     octet ( Integer                       ) ->  this Bitstring raise OutOfRange;
axioms
/* Bitstring starts at index 0 */
/* Definition of operators with the same names as String operators*/
     for all b in Bit ( /*b is bit in string*/
     for all s,s1,s2,s3 in Bitstring (
     for all i,j in Integer (
/* constructors are ''B, mkstring, and "//" */
/* equalities between constructor terms */
     s // ''B          == s;
     ''B// s           == s;
     (s1 // s2) // s3== s1 // (s2 // s3);
/* definition of length by applying it to all constructors */
     <<type Bitstring>>length(''B)               == 0;
     <<type Bitstring >>length(mkstring(b))      == 1;
     <<type Bitstring >>length(s1 // s2)         == length(s1) + length(s2);
     make(s)                                     == mkstring(s);
/* definition of extract by applying it to all constructors,
        with error cases handled separately */
     extract(mkstring(b),0)                      == b;
     i < length(s1)      ==> extract(s1 // s2,i) == extract(s1,i);
     i >= length(s1)     ==> extract(s1 // s2,i) == extract(s2,i-length(s1));
     i<0 or i=>length(s)  ==> extract(s,i)        == raise InvalidIndex;
/* definition of first and last by other operations */
     first(s)   == extract(s,0);
     last(s)    == extract(s,length(s)-1);
/* definition of substring(s,i,j) by induction on j,
        error cases handled separately */
     i>=0 and i < length(s)            ==>
          substring(s,i,0)  == ''B;
/* */
     i>=0 and j>0 and i+j<=length(s) ==>
          substring(s,i,j)  == substring(s,i,j-1) // mkstring(extract(s,i+j));
/* */
     i<0 or j<0 or i+j>length(s)       ==>
          substring(s,i,j)  == raise InvalidIndex;
/* */
/* definition of modify by other operations */
     modify(s,i,b)== substring(s,0,i) // mkstring(b) // substring(s,i+1,length(s)-i);
/* definition of remove */
     remove(s,i,j)   == substring(s,0,i) // substring(s,i+j,length(s)-i-j);
     )));
/*end of definition of string operators indexed from zero*/
/* */
/* Definition of ''H and 'x'H in terms of ''B, 'xxxx'B for Bitstring*/
     <<type Bitstring>>''H  == ''B;
     <<type Bitstring>>'0'H == '0000'B;
     <<type Bitstring>>'1'H == '0001'B;
     <<type Bitstring>>'2'H == '0010'B;
     <<type Bitstring>>'3'H == '0011'B;
     <<type Bitstring>>'4'H == '0100'B;
     <<type Bitstring>>'5'H == '0101'B;
     <<type Bitstring>>'6'H == '0110'B;
     <<type Bitstring>>'7'H == '0111'B;
     <<type Bitstring>>'8'H == '1000'B;
     <<type Bitstring>>'9'H == '1001'B;
     <<type Bitstring>>'A'H == '1010'B;
     <<type Bitstring>>'B'H == '1011'B;
     <<type Bitstring>>'C'H == '1100'B;
     <<type Bitstring>>'D'H == '1101'B;
     <<type Bitstring>>'E'H == '1110'B;
     <<type Bitstring>>'F'H == '1111'B;
/* */
/* Definition of Bitstring specific operators*/
     <<type Bitstring>>mkstring(0)          == '0'B;
     <<type Bitstring>>mkstring(1)          == '1'B;
/* */
     for all s, s1, s2, s3 in Bitstring (
     s  =  s                                == true;
     s1 =  s2                               == s2 = s1;
     s1 /= s2                               == not ( s1 = s2 );
     s1 =  s2 == true            ==> s1       == s2;
     ((s1 = s2) and (s2 = s3))  ==> s1 = s3  == true;
     ((s1 = s2) and (s2 /= s3)) ==> s1 = s3  == false;
/* */
     for all b, b1, b2 in Bit (
     not(''B)                     == ''B;
     not(mkstring(b) // s)        == mkstring( not(b) ) // not(s);
/* definition of or */
/* The length of or-ing two strings is the maximal length of both strings */
     ''B or ''B                   == ''B;
     length(s) > 0 ==> ''B or s == mkstring(0) or s;
     s1 or s2                     == s2 or s1;
     (b1 or b2) // (s1 or s2)     ==(mkstring(b1) // s1) or (mkstring(b2) // s2);
/* */
/* definition of remaining operators based on "or" and "not" */
     s1 and s2                    == not (not s1 or not s2);
     s1 xor s2                    == (s1 or s2) and not(s1 and s2);
     s1 => s2                     == not (s1 and s2);
     ));
/* */
/*Definition of 'xxxxx'B literals */
     for all s in Bitstring (
     for all b in Bit (
     for all i in Integer (
     <<type Bitstring>>num (''B)            == 0;
     <<type Bitstring>>bitstring (0)      == '0'B;
     <<type Bitstring>>bitstring (1)      == '1'B;
     num (s // mkstring (b))              == num (b) + 2 * num (s);
     i > 1                ==> bitstring (i)  == bitstring (i / 2) // bitstring (i mod 2);
     i >= 0 and i <= 255  ==> octet (i)   == bitstring (i) or '00000000'B;
     i < 0                ==> bitstring (i)  == raise OutOfRange;
     i < 0 or i > 255   ==> octet (i)   == raise OutOfRange;
     )))
```

```
/*Definition of 'xxxxx'H literals */
     for all b1,b2,b3,h1,h2,h3 in bs nameclass (
     for all bs1, bs2, bs3, hs1, hs2, hs3 in Charstring (
     spelling(b1) = '''' // bs1 // '''B',
     spelling(b2) = '''' // bs2 // '''B',
     bs1 /= bs2                             ==> b1 = b2    == false;
/* */
     spelling(h1) = '''' // hs1 // '''H',
     spelling(h2) = '''' // hs2 // '''H',
     hs1 /= hs2 ==> h1 = h2 == false;
     spelling(b1) = '''' // bs1 // '''B',
     spelling(b2) = '''' // bs2 // '''B',
     spelling(b3) = '''' // bs1 // bs2 // '''B',
     spelling(h1) = '''' // hs1 // '''H',
     spelling(h2) = '''' // hs2 // '''H',
     spelling(h3) = '''' // hs1 // hs2 // '''H',
     length(bs1) = 4,
     length(hs1) = 1,
     length(hs2) > 0,
     length(bs2) = 4 * length(hs2),
     h1 = b1                                ==> h3 = b3    == h2 = b2;
/* */
/* connection to the String generator */
     for all b in Bit literals (
     spelling(b1) = '''' // bs1 // bs2 // '''B',
     spelling(b2) = '''' // bs2 // '''B',
     spelling(b)  = bs1                     ==> b1         == mkstring(b) // b2;
     )));
endvalue type Bitstring;
/*
```

D.3.15 ASN.1 Octet and Octetstring sorts

D.3.15.1 Definition

```
*/
syntype Octet = Bitstring size (8)
endsyntype Octet;
/* */
value type Octetstring
     inherits String < Octet > ( ''B = emptystring )
     adding
     operators
     os in nameclass
          '''' ( ((('0' or '1')8)*'''B') or (((('0':'9') or ('A':'F'))2)*'''H') )
           -> this Octetstring;
     bitstring   ( this Octetstring ) -> Bitstring;
     octetstring ( Bitstring ) -> this Octetstring;
axioms
     for all b,b1,b2 in Bitstring (
     for all s in Octetstring (
     for all o in Octet(
     <<type Octetstring>> bitstring(''B)            == ''B;
     <<type Octetstring>> octetstring(''B)          == ''B;
     bitstring( mkstring(o) // s )                  == o // bitstring(s);
/* */
     length(b1) > 0,
     length(b1) < 8,
     b2 == b1 or '00000000'B==> octetstring(b1)  == mkstring(b2);
/* */
     b == b1 // b2,
     length(b1) == 8            ==> octetstring(b)== mkstring(b1) // octetstring(b2);
     )));
/* */
     for all b1, b2 in Bitstring (
     for all o1, o2 in os nameclass (
     spelling( o1 ) = spelling( b1 ),
     spelling( o2 ) = spelling( b2 ) ==> o1 = o2 == b1 = b2
     ));
endvalue type Octetstring;
/*
```

D.3.16 Predefined Exceptions

```
*/
exception
     OutOfRange,          /* A range check has failed. */
     InvalidReference,    /* Null was used incorrectly. Wrong Pid for this signal. */
     NoMatchingAnswer,    /* No answer matched in a decision without else part. */
     UndefinedVariable,/* A variable was used that is “undefined”. */
     UndefinedField,      /* An undefined field of a choice or struct was accessed. */
     InvalidIndex,     /* A String or Array was accessed with an incorrect index. */
     DivisionByZero;      /* An Integer or Real division by zero was attempted. */
     Empty;               /* No element could be returned. */
/* */
endpackage Predefined;
```

ANNEX E

Reserved for examples

ANNEX F

Formal Definition

Further study needed for SDL-2000.

APPENDIX I

Status of Z.100, related documents and Recommendations

This appendix contains a list of the status of SDL related documents issued by ITU-T. The list includes all parts of Z.100, Z.105, Z.106, Z.107, Z.109 and any related methodology documents. It also lists other relevant documents such as Z.110.

This list shall be updated by appropriate means (for example a Corrigendum) whenever changes to SDL are agreed and new documents approved.

SDL-2000 is defined by the following Recommendations approved by ITU-T Study Group 10 on 19 November 1999.

Z.100 (11/99) – Specification and Description Language (SDL).

Annex A to Z.100 – Index of Non-terminals

Annex D to Z.100 – SDL Predefined Data

Annex F is planned for November 2000.

No specific plans at the time of approval for Annexes B, C and E.

Supplement 1 to Z.100 (05/97): SDL+ Methodology. *Update for SDL-2000 in progress.*

Z.105 (11/99) SDL Combined with ASN.1 Modules.

Z.106 (10/96) Common Interchange Format for SDL. *Update for SDL-2000 in progress.*

Z.107 (11/99) SDL with Embedded ASN.1.

Z.109 (11/99) SDL Combined with UML.

Z.110 (10/96) Criteria for the use of Formal Description Techniques by ITU-T. *Update planned for November 2000.*

Further information on SDL including information on books and other publications is available via: http://www.sdl-forum.org.

APPENDIX II

Guidelines for the maintenance of SDL

II.1 Maintenance of SDL

This appendix describes the terminology and rules for maintenance of SDL agreed at the Study Group 10 meeting in November 1993, and the associated "change request procedure".

II.1.1 Terminology

a) An *error* is an internal inconsistency within Z.100.

b) A *textual correction* is a change to text or diagrams of Z.100 that corrects clerical or typographical errors.

c) An *open item* is a concern identified but not resolved. An open item may be identified either by a Change Request, or by agreement of the Study Group or Working Party.

d) A *deficiency* is an issue identified where the semantics of SDL are not (clearly) defined by Z.100.

e) A *clarification* is a change to the text or diagrams of Z.100 that clarifies previous text or diagrams which could be ambiguously understood without the clarification. The clarification should attempt to make Z.100 correspond to the semantics of SDL as understood by the Study Group or Working Party.

f) A *modification* is a change to the text or diagrams of Z.100 that changes the semantics of SDL.

g) A *decommitted feature* is a feature of SDL that is to be removed from SDL in the next revision of Z.100.

h) An *extension* is a new feature, which must not change the semantics of features defined in Z.100.

II.1.2 Rules for Maintenance

In the following text references to Z.100 shall be considered to include Annexes, Appendices, and Supplements, as well as any Addendum, or Amendment or Corrigendum or Implementor's Guides, and the same texts for Z.105, Z.106, Z.107 and Z.109.

a) When an error or deficiency is detected in Z.100, it must be corrected or clarified. The correction of an error should imply as small a change as possible. Error corrections and clarifications will be put into the Master list of Changes for Z.100 and come into effect immediately.

b) Except for error corrections and resolution of open items from the previous study period, modifications and extensions should only be considered as the result of a request for change that is supported by a substantial user community. A request for change should be followed by investigation by the Study Group or Working Party in collaboration with representatives of the user group, so that the need and benefit are clearly established and it is certain that an existing feature of SDL is unsuitable.

c) Modifications and extensions not resulting from error correction shall be widely publicised and the views of users and toolmakers canvassed before the change is adopted. Unless there are special circumstances requiring such changes to be implemented as soon as possible, such changes will not be recommended until Z.100 is revised.

d) Until a revised Z.100 is published a Master list of Changes to Z.100 will be maintained covering Z.100 and all annexes except the formal definition. Appendices, Addenda, Corrigenda, Implementor's guides or Supplements will be issued as decided by the Study

Group. To ensure effective distribution of the Master list of Changes to Z.100, it will be published as COM Reports and by appropriate electronic means.

e) For deficiencies in Z.100 the formal definition should be consulted. This may lead to either a clarification or correction that is recorded in the Master list of changes to Z.100.

II.1.3 Change request procedure

The change request procedure is designed to enable SDL users from within and outside ITU-T to ask questions about the precise meaning of Z.100, make suggestions for changes to SDL or Z.100, and to provide feedback on proposed changes to SDL. The SDL experts' group shall publish proposed changes to SDL before they are implemented.

Requests for changes should either use the Change Request Form (see below) or provide the information listed by the form. The kind of request should be clearly indicated (error correction, clarification, simplification, extension, modification or decommitted feature). It is also important that for any change other than an error correction, the amount of user support for the request is indicated.

Meetings of the ITU-T study group responsible for Z.100 should treat all change requests. For corrections or clarifications the changes may be put on the list of corrections without consulting users. Otherwise a list of open items is compiled. The information should be distributed to users:

- as ITU-T white contribution reports;
- by electronic mail to SDL mailing lists (such as ITU-T informal list, and sdlnews@sdl-forum.org);
- other means as agreed by the Study Group 10 experts.

Study group experts should determine the level of support and opposition for each change and evaluate reactions from users. A change will only be put on the accepted list of changes if there is substantial user support and no serious objections to the proposal from more than just a few users. Finally all accepted changes will be incorporated into a revised Z.100. Users should be aware that until changes have been incorporated and approved by the study group responsible for Z.100 they are not recommended by ITU-T.

Change Request Form

Please fill in the following details		
Character of change:	☐ error correction ☐ simplification ☐ modification	☐ clarification ☐ extension ☐ decommission
Short summary of change request		
Short justification of the change request		
Is this view shared in your organization	☐ yes	☐ no
Have you consulted other users	☐ yes	☐ no
How many users do you represent?	☐ 1-5 ☐ 11-100	☐ 6-10 ☐ over 100
Your name and address		

please attach further sheets with details if necessary

SDL (Z.100) Rapporteur, c/o ITU-T, Place des Nations, CH-1211, Geneva 20, Switzerland. Fax: +41 22 730 5853, e-mail: SDL.rapporteur@itu.int

APPENDIX III

Systematic conversion of SDL-92 to SDL-2000

Although not all SDL-92 specifications can be automatically converted into SDL-2000, a simple transformation should be sufficient in many cases. The following procedure assumes a conversion processing SDL/PR; a similar procedure transforming SDL/GR is also possible:

1) Correct spelling with regard to case and new keywords:
   a) Replace all keywords with the corresponding <lowercase keyword> (or <uppercase keyword>);
   b) Replace all <word>s containing <national> characters as defined in Z.100 (03/93) with a unique <word>;
   c) Replace all <name>s with their lowercase equivalent;
   d) If <name>s conflict with <lowercase keyword>s, replace the first character with an uppercase character.

   In many cases, a more relaxed procedure is possible, for example by always using the spelling of the <name> defining occurrence of its corresponding <identifier>. This results in a semantical change only if the name of a state is changed, and that name is used in a <state expression>, as introduced in Addendum 1 of SDL-92.
2) Replace all <qualifier>s with the corresponding <qualifier> of SDL-2000 (that is, the list of path items is always enclosed in the <composite special>s <qualifier begin sign> and <qualifier end sign>).
3) Transform all usage of the keywords fpar and return in <agent formal parameters>, <procedure formal parameters>, <procedure return>, <macro formal parameter>, <formal operation parameters>, and <procedure signature> to the corresponding SDL-2000 syntax.
4) Replace all signal routes with nodelay <channel definition>s.
5) In each block with no signal routes or channels, add gates to each process listing all signals that are sent or received by this process. Alternatively, add implicit channels according to the model for implicit signalroutes of SDL-92. Specifications relying on implicit channels as introduced by Addendum 1 must also add gates to the respective blocks.
6) Replace all occurrences of block partitioning. SDL-92 did not specify how a consistent subset was selected, so this step might require external knowledge. A conversion assuming that the substructure should be always selected would probably reflect the typical use of SDL-92.
   a) Move all blocks in the substructure directly into the container block.
   b) If there are conflicts in entities of the block and entities in the substructure, rename one of the entities to a unique name.
   c) Adjust all identifiers for entities in nested blocks to use the new qualifier.
7) Replace all output actions using via all with a list of output actions. If the <via path> was a channel between block instance sets, no automatic transformation is possible.
8) Replace service and service types with composite state and composite state types, respectively. If the services have overlapping input actions (even though their valid input sets were disjoint), the one of the duplicate transitions must be removed. Remove all signal routes between services; output referring to these signal routes should refer to gates of the process type. Replace <stop> with <return>. Timers, exported procedures and exported variables of a service must be defined in agent.
9) Replace all data type definitions involving generator transformations by the equivalent definitions using parameterized types.

10) Transformation of data axioms is not possible automatically, but there have been only a few users that have defined their own data types axiomatically. However, the following cases can be transformed easily:

   a) Expressions of predefined data remain valid including the use String, Array and PowerSet, after case adjustments in the spelling of their types.

   b) A newtype definition with literals (and no axiomatically defined operators) can be converted into a <value data type definition> with <literal list>.

   c) A newtype definition with structure property can be converted into a <value data type definition> with <structure definition>.

If an SDL-92 specification uses constructs that cannot be automatically converted into equivalent SDL-2000 constructs, a careful inspection of this specification will be necessary if it needs to conform to this Recommendation.

18355

Printed in Switzerland
Geneva, 2000