I n t e r n a t i o n a l   T e l e c o m m u n i c a t i o n   U n i o n

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Z.101

(10/2019)

SERIES Z: LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

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

Specification and Description Language – Basic SDL 2010

Recommendation ITU-T Z.101

ITU-T

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 (MSC)	Z.120–Z.129
User Requirements Notation (URN)	Z.150–Z.159
Testing and Test Control Notation (TTCN)	Z.160–Z.179
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.349
Data-oriented human-machine interfaces	Z.350–Z.359
Human-machine interfaces for the management of telecommunications networks	Z.360–Z.379
QUALITY	
Quality of telecommunication software	Z.400–Z.409
Quality aspects of protocol-related Recommendations	Z.450–Z.459
METHODS	
Methods for validation and testing	Z.500–Z.519
MIDDLEWARE	
Processing environment architectures	Z.600–Z.609

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

Recommendation ITU-T Z.101

Specification and Description Language – Basic SDL-2010

Summary

Recommendation ITU-T Z.101 defines the basic features of the Specification and Description Language. Together with Recommendations ITU-T Z.100, ITU-T Z.102, ITU-T Z.103, ITU-T Z.104, ITU-T Z.105, ITU-T Z.106 and ITU-T Z.107, this Recommendation is part of a reference manual for the language. The language defined in this document covers the essential features of the language, which is further defined in other Recommendations in the ITU-T Z.100 series.

Coverage

The Specification and Description Language 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. A distinctive feature of the Specification and Description Language is the graphical representation. This Recommendation covers the main features of the language such as agent (block, process) type diagrams, agent diagrams for structures with channels, diagrams for extended finite state machines and the associated semantics for these basic features. The concrete grammar given is the graphical representation. The alternative textual programming representation is given in Recommendation ITU-T Z.106. The concrete grammar in this Recommendation gives a canonical syntax, which is extended in Recommendation ITU-T Z.103 to a syntax that is easier to use. The basic part of the language given in this Recommendation does not include the details of expressions, data definitions and action language, which is defined in Recommendation ITU-T Z.104 and for object-oriented data in ITU-T Z.107. The features of the language defined in Recommendation ITU-T Z.102 make the language more comprehensive.

Applications

The Specification and Description Language is applicable within standard bodies and industry. The main application areas for which The Specification and Description Language has been designed are stated in ITU-T Z.100, but the language is generally suitable for describing reactive systems. The range of application is from requirement description to implementation. The features of the language defined in ITU-T Z.101 allow basic models to be defined and provide a basis for other features defined in other Recommendations in the ITU-T Z.100 series.

History

Edition	Recommendation	Approval	Study Group	Unique ID*
1.0	ITU-T Z.101	2011-12-22	17	11.1002/1000/11388
2.0	ITU-T Z.101	2016-04-29	17	11.1002/1000/12847
3.0	ITU-T Z.101	2019-10-14	17	11.1002/1000/14052

Keywords

Actions, agents, Basic SDL-2010, canonical syntax, channels, data, essential features, extended finite state machine, SDL-2010, Specification and Description Language, structure.

* To access the Recommendation, type the URL http://handle.itu.int/ in the address field of your web browser, followed by the Recommendation's unique ID. For example, http://handle.itu.int/11.1002/1000/11830-en.

FOREWORD

The International Telecommunication Union (ITU) is the United Nations specialized agency in the field of telecommunications, information and communication technologies (ICTs). The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of ITU. ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.

The World Telecommunication Standardization Assembly (WTSA), which meets every four years, establishes the topics for study by the ITU-T study groups which, in turn, produce Recommendations on these topics.

The approval of ITU-T Recommendations is covered by the procedure laid down in WTSA Resolution 1.

In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC.

NOTE

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

Compliance with this Recommendation is voluntary. However, the Recommendation may contain certain mandatory provisions (to ensure, e.g., interoperability or applicability) and compliance with the Recommendation is achieved when all of these mandatory provisions are met. The words "shall" or some other obligatory language such as "must" and the negative equivalents are used to express requirements. The use of such words does not suggest that compliance with the Recommendation is required of any party.

INTELLECTUAL PROPERTY RIGHTS

ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. ITU takes no position concerning the evidence, validity or applicability of claimed Intellectual Property Rights, whether asserted by ITU members or others outside of the Recommendation development process.

As of the date of approval of this Recommendation, ITU had not received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementers are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database at http://www.itu.int/ITU-T/ipr/.

Table of Contents

Introduction

This Recommendation is part of the ITU-T Z.100 to ITU-T Z.107 series of Recommendations that give the complete language reference manual for SDL-2010. The text of this Recommendation is stable. For more details see Recommendation ITU-T Z.100.

Recommendation ITU-T Z.101

Specification and Description Language – Basic SDL-2010

1 Scope and objective

This Recommendation defines the basic features of the Specification and Description Language. The language defined in this document covers the essential features of the language, which is further defined in other Recommendations in the ITU-T Z.100 series. Together with Recommendations [ITU-T Z.100], [ITU-T Z.102], [ITU-T Z.103], [ITU-T Z.104], [ITU-T Z.105], [ITU-T Z.106] and [ITU-T Z.107], this Recommendation forms a reference manual for the language.

1.1 Objective

The objective of this Recommendation is to define the basic features of the Specification and Description Language in a canonical concrete syntax. The language defined in this Recommendation is a strict subset of SDL-2010.

The main features not included in this Recommendation are macros, specialization, context parameters, remote procedures and remote variables, state aggregation, priority input, enabling conditions, spontaneous transitions, exceptions, compound statements (other than as task bodies), object data types, synonyms and generic system definition. Where there is choice of syntax for the same abstract grammar the graphical syntax has been chosen, so that, for example, procedure diagram is used rather than procedure definition.

A specification in SDL-2010 starts with an instantiation of a system type. For that reason, Basic SDL-2010 does not include a system diagram. In Comprehensive SDL-2010 such a diagram is considered shorthand for the instantiation of a system type. Similarly, block diagrams and process diagrams are not included, only diagrams for agent types and other types such as composite state and procedure.

The canonical syntax is chosen to be the syntax supported by most tools in the cases where that is simply an alternative to the syntax introduced in SDL-2000 (for example, the use of the keyword **returns** rather than <result sign>).

1.2 Application

This Recommendation is part of the reference manual for the Specification and Description Language. The part of the language defined by this Recommendation does not usually include shorthand notation or *Model* clauses, so that a model written using only this part of SDL-2010 is not as concise or as readable as one using the full language. The part of the language defined in this Recommendation is mainly applicable if a model is required that is limited in the language features it uses, and it is intended that the model is presented in a concrete form that closely matches the abstract syntax.

In some cases a *Model* clause or shorthand has been included, because this was considered the most practical way to describe a feature. One example is the *Model* clause for expression that explains how the concrete infix syntax maps to operation application.

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; 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. The reference to a document within this Recommendation does not give it, as a stand-alone document, the status of a Recommendation.

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

[ITU-T Z.100] Recommendation ITU-T Z.100 (2019), *Specification and Description Language – Overview of SDL-2010*.

[ITU-T Z.102] Recommendation ITU-T Z.102 (2019), *Specification and Description Language – Comprehensive SDL-2010*.

[ITU-T Z.103] Recommendation ITU-T Z.103 (2019), *Specification and Description Language – Shorthand notation and annotation in SDL-2010*.

[ITU-T Z.104] Recommendation ITU-T Z.104 (2019), *Specification and Description Language – Data and action language in SDL-2010*.

[ITU-T Z.105] Recommendation ITU-T Z.105 (2019), *Specification and Description Language – SDL-2010 combined with ASN.1 modules*.

[ITU-T Z.106] Recommendation ITU-T Z.106 (2019), *Specification and Description Language – Common interchange format for SDL-2010*.

[ITU-T Z.107] Recommendation ITU-T Z.107 (2019), *Specification and Description Language – Object-oriented data in SDL-2010*.

[ITU-T Z.111] Recommendation ITU-T Z.111 (2016), *Notations and guidelines for the definition of ITU-T languages*.

3 Definitions

3.1 Terms defined elsewhere

This Recommendation uses the following terms defined elsewhere:

The definitions of [ITU-T Z.100] apply.

3.2 Terms defined in this Recommendation

None.

4 Abbreviations and acronyms

This Recommendation uses the following abbreviations and acronyms:

The abbreviations and acronyms defined in [ITU-T Z.100] apply.

5 Conventions

The conventions defined in [ITU-T Z.100] apply; these include the conventions defined in [ITU-T Z.111].

Each abstract syntax item in this Recommendation is contained by at least one other abstract syntax item in this Recommendation except:

Sdl-specification	which is the container for all other items.

The concrete syntax rules defined in this Recommendation are all used by other syntax rules in this Recommendation except the following:

<sdl specification>	The starting rule for the concrete syntax.
<lexical unit>	A rule to collect the lexical units.
<flow line symbol>	Implicit for ***is followed by***.
<simple expression>	Only used in combination with semantic subcategories.
<type expression>	Only used in combination with semantic subcategories.

6 General rules

General rules cover: lexical units; the use of a semicolon with comment as a terminator; commonly used symbols; the visibility, resolution and use of names and identifiers; the use of frames and page numbers.

6.1 Lexical rules

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

```
<lexical unit> ::=
                        <name>
            |           <integer name>
            |           <real name>
            |           <character string>
            |           <hex string>
            |           <bit string>
            |           <note>
            |           <comment body>
            |           <composite special>
            |           <special>
            |           <semicolon>
            |           <other character>
            |           <quoted operation name>
            |           <keyword>
```

NOTE 1 – A lexical distinction is made between a <name>, an <integer name> and a <real name> whereas in SDL-2000 these are all <name>.

NOTE 2 – The alternatives of <composite special> (such as <result sign>, <range sign>), <special> (such as <asterisk>, and including the alternatives of <other special> such as <hyphen>) are used as terminal symbols in the *Concrete grammar*. The alternatives of <keyword> are also used in the *Concrete grammar*. Other lexical rules (such as <word> and <other character> that are not alternatives of <lexical unit> are used only in the lexical rules.

NOTE 3 – As a lexical unit, <other character> only occurs in the *Concrete grammar* as part of annotations such as in <comment>.

```
<name> ::=
                        <underline>+ <word> {<underline>+ <word>}* <underline>*
            |           <word> <underline>+ [ <word>{<underline>+ <word>}* <underline>* ]
            |           <decimal digit>* <letter> <alphanumeric>*
```

If a <letter> sequence is defined as a <keyword>, it is not allowed as a <name>. For example, **block** is not allowed as a name. This resolves the lexical ambiguity that otherwise would exist in this case in a way that is independent of the use of the lexical unit in the *Concrete grammar*.

```
<integer name> ::=
                        <decimal digit>+
```

<real name> ::=
<integer name> <full stop> <integer name>
[{ e | E } [<hyphen> | <plus sign>] <integer name>]

<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>
| <quotation mark> <equals sign> <quotation mark>
| <quotation mark> <not equals sign> <quotation mark>

<infix operation name> ::=
or | **xor** | **and** | **in** | **mod** | **rem**
| <plus sign> | <hyphen>
| <asterisk> | <solidus>
| <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>
| <semicolon>
| <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>
                    |    <solidus>
                    |    <asterisk>+ <not asterisk or solidus>
                    |    <number sign>
                    |    <other special>
                    |    <apostrophe> }*
```

A <note> is lexical unit that is a form of annotation. The <note> has no formal semantic meaning.

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

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

<comment body> ::=
                    <solidus> <number sign>   <comment text> <number sign>+ <solidus>
```

NOTE 4 – A <comment body> is used in contexts where annotation is needed and a <comment> is not possible because there is no <semicolon> at this point. This occurs before <left curly bracket> in several places such as in <data type definition>. A <comment body> is also allowed within a <comment> so that a <semicolon> included in the <comment body> does not end the <comment>.

```
<comment text> ::=
               {         <general text character>
                    |    <semicolon>
                    |    <solidus>
                    |    <asterisk>
                    |    <number sign>+ <not number or solidus>
                    |    <other special>
                    |    <apostrophe> }*

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

<composite special> ::=
                    <result sign>
               |    <range 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>

<range sign> ::=
                    <full stop> <full stop>

<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>     |     <number sign>     |     <other special>

<other special> ::=
                    <exclamation mark>
                |   <left parenthesis>        |   <right parenthesis>
                |   <plus sign>               |   <comma>                 |   <hyphen>
                |   <full stop>               |   <colon>
                |   <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**
| **all** | **aggregation** | **alternative**
| **and** | **any** | **as**
| **association** | **atleast** | **block**
| **break** | **call** | **channel**
| **choice** | **comment** | **composition**
| **connect** | **connection** | **constants**
| **continue** | **create** | **dcl**
| **decision** | **decode** | **default**
| **else** | **encode** | **endalternative**
| **endblock** | **endchannel** | **endconnection**
| **enddecision** | **endexceptionhandler** | **endinterface**
| **endmacro** | **endmethod** | **endobject**
| **endnewtype** | **endoperator** | **endpackage**
| **endprocedure** | **endprocess** | **endselect**
| **endstate** | **endsubstructure** | **endsyntype**
| **endsystem** | **endtype** | **endvalue**
| **env** | **exception** | **exceptionhandler**
| **export** | **exported** | **external**
| **fi** | **finalized** | **fpar**
| **from** | **gate** | **handle**
| **if** | **import** | **in**
| **inherits** | **input** | **interface**
| **join** | **literals** | **loop**
| **macro** | **macrodefinition** | **macroid**
| **method** | **methods** | **mod**
| **nameclass** | **newtype** | **nextstate**
| **nodelay** | **none** | **not**
| **now** | **object** | **offspring**
| **onexception** | **operator** | **operators**
| **optional** | **or** | **ordered**
| **own** | **out** | **output**
| **package** | **parent** | **part**
| **priority** | **private** | **procedure**
| **protected** | **process** | **provided**
| **public** | **raise** | **redefined**
| **ref** | **referenced** | **rem**
| **remote** | **reset** | **return**
| **returns** | **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> ::=

Some keywords (such as **exceptionhandler** and **object**) are not used in SDL-2010 but were valid keywords in SDL-2000 and therefore are not allowed as names to avoid issues for tools that support earlier versions of the language.

The keywords **ref** and **own** are introduced for object-oriented data and are not used in this Recommendation, [ITU-T Z.102], [ITU-T Z.103], [ITU-T Z.104] or [ITU-T Z.105].

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 (IRV) of the International Reference

Alphabet [ITU-T T.50]). The lexical unit <space> represents the ITU-T T.50 SPACE character (acronym SP), which (for obvious reasons) it is not possible to show.

IRV delete characters are completely ignored.

If an extended character set is used, the printing characters that are not defined by IRV are permitted to appear freely in a <character string> in a <comment> or within a <note>. A printing character of an extended character set that corresponds to an IRV <letter> is equivalent to the IRV <letter>. Similarly, printing character of an extended character set that corresponds to an IRV <decimal digit>, <special>, <other special> or <other character> is equivalent to the IRV <decimal digit>, <special>, <other special> or <other character>, respectively. A printing character of an extended character set that represents a letter in some script and does not correspond to an IRV <letter> is allowed to be used as a <letter>. Characters of an extended character set are treated in the order they occur in the model source, which possibly does not correspond to the apparent printing order depending on how characters in the script are printed (such as right to left, left to right or in combination).

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.

NOTE 5 – A <name> ending in an <underline> followed by a <space> or control character has to be written with an extra <underline>. For example, the definition an integer variable with the name "`IV_`" is written "**`dcl`** `IV__ Integer;`".

A (non-space) control character is allowed wherever a <space> is allowed, 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 character string expression containing control characters, the <concatenation sign> operator and the literals for control characters have to be used. All spaces in a character string are significant: a sequence of spaces is not treated as one space in a character string.

It is allowed to insert any number of <space>s 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 and the *Concrete grammar*, the lowercase <keyword> as a terminal denotes that the uppercase <keyword> with the same spelling is allowed at the same place. For example, the keyword

default

represents the lexical alternatives

{ **default** | **DEFAULT** }

NOTE 6 – Boldface lower case is used for keywords within this Recommendation. Distinguishing by font attributes is not a mandatory requirement, but is useful to the readers of a specification.

NOTE 7 – Versions of the language before SDL-2000 were not case sensitive. Keywords of the language could be in mixed case, and different occurrences of the same name could have a different case mix. Although it is possible that some tools support a mode where they are case insensitive, a model that is not case correct in the spelling of keywords and inconsistent in the case usage for a name is not valid. The model is required to be case correct according to SDL-2010 lexical rules.

The first character that is not part of the <lexical unit> according to the syntax specified above terminates the <lexical unit>.

NOTE 8 – If a <lexical unit> is possible either a <name> and a <keyword>, it is a <keyword> (see the constraint on <name> above).

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 "or"(a, b), which means the same as (a **or** b).

Special lexical rules apply within macros (see [ITU-T Z.102]).

6.2 End terminator and comment

A semicolon is used in many places as a terminator. In most contexts it is possible to precede it by a comment.

Concrete grammar

```
<end> ::=
                    [<comment>] <semicolon>

<comment> ::=
                    comment {     <name or number>
                              |   <string name>
                              |   <note>
                              |   <comment body>
                              |   <composite special>
                              |   <special>
                              |   <other special>
                              |   <other character>
                              |   <keyword>    }*
```

See clause 6.6 for <name or number> and <string name>.

The lexical elements in a <comment> are treated as annotation and do not have any formal meaning and are ignored when any transformations are applied.

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

NOTE – It is not easy to state this last rule by syntax, because it applies to all non-terminals used by the above non-terminals, but they are also used in other non-terminals where comment is allowed.

6.3 Empty clause

This clause is intentionally left blank.

6.4 Solid association symbol

The solid association symbol is used in several places with other graphical symbols, for example, between a state symbol and an <input area> of the state.

Concrete grammar

```
<solid association symbol> ::=
                    ______________
```

A <solid association symbol> is a line symbol (see *Concrete grammar* in clause 5.3.2 of [ITU-T Z.100]).

6.5 The metasymbol ***is followed by*** and flow line symbols

Concrete grammar

The metasymbol ***is followed by*** is used in the concrete syntax between a graphical non-terminal symbol as the left hand argument and right hand argument that is a group of syntactic elements within curly brackets or a single syntactic element. The representation is that there is a <flow line symbol> between the left hand argument and the right hand argument. The logical direction of flow is from the left hand argument to the right hand argument.

<flow line symbol> ::=
    <flow line symbol with arrowhead>

<flow line symbol with arrowhead> ::=
    $\longrightarrow$

The non-terminal <flow line symbol> is never used explicitly in the concrete syntax rules, and therefore only occurs in diagrams as a representation of ***is followed by*** in a syntax rule.

A <flow line symbol> is a line symbol (see *Concrete grammar* in clause 5.3.2 of [ITU-T Z.100]).

A <flow line symbol> by default flows from the middle bottom of the left hand argument to the middle top of the right hand argument of ***is followed by***. A number of straight-line segments are allowed in the flow line, so two arguments of ***is followed by*** are able to be below or above or to the left or the right of each other in a diagram.

6.6 Names and identifiers, name resolution and visibility

The name of an item is established by the definition of the item. It is allowed to use the same name for different items, but the identity of the item includes the context of the definition, the entity kind and in some cases other attributes such as the signature in the case of an operation. The definition context is the path to the item definition from an outer level package or the system. When an item is used, the identity of the item is usually established through an identifier, which includes a qualifier that gives the path to the item definition. However, if the complete qualifier were given in the concrete syntax for every identifier, all these qualifiers would obscure the specification and make it tedious to write. For this reason it is allowed to omit the qualifier or part of the qualifier in the concrete syntax of an identifier if the item is still unambiguously established. In most cases a name is sufficient. This is a shorthand notation that therefore should in principle be in [ITU-T Z.103], but because of the importance in making specifications readable this shorthand is described here.

In the context of most definitions only the concrete syntax for an alphanumeric <name> is allowed. For most identifiers therefore only a <name> is allowed as the name part. In some cases (such as literals and operation names) other forms such as numbers and <quoted operation name>s are allowed. The syntax for <identifier> therefore includes these alternatives, but the name of an agent (for example) has to be defined in the form <name>.

If only the features defined in the basic language defined in this Recommendation are used, name resolution is fairly simple. The additional features defined in [ITU-T Z.102], [ITU-T Z.103] and [ITU-T Z.104] increase the complexity of name resolution. This clause includes the parts of name resolution relevant to [ITU-T Z.102], [ITU-T Z.103] and [ITU-T Z.104], to avoid confusion by introducing name resolution incrementally in the different places.

To use a name in an identifier it also has to be visible. There are scope units that restrict the visibility of names of items defined within the scope unit. This avoids every name having global visibility, so that re-use of names is possible in different places without the need to qualify the corresponding 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*
	|	*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*
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*

The *Path-item* list of a *Qualifier* shall give the full path to the identified entity. For the system and any package not contained in another package the full path is an empty *Path-item* list.

Each *Name* has a different *Token*.

Concrete grammar

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

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

If the <qualifier> is omitted or the <path item> list does not give the full path to the named item, the *Qualifier* is determined by name resolution and the *Model* given below is applied.

NOTE 1 – If the full path were given for every identifier, in most cases the <identifier> would be longer than necessary and the description would be unreadable. For each use of the *Identifier*, there is usually a minimal form where the <qualifier> is omitted or as many leftmost <path item> elements are omitted as possible. In the minimal form, for each <path item> that is needed the <scope unit kind> is omitted wherever possible (see below). As a guideline this minimal form (usually just the <name or number>) should be used whenever possible, and is considered the canonical concrete syntax form and adding a redundant <path item> or <scope unit kind> is considered annotation.

<name or number> ::=
<name>
| <integer name>
| <real name>
| <quoted operation name>
| <string name>

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

A <name or number> of <identifier> represents the *Name* of an *Identifier* in the *Abstract syntax*. The *Token* for the *Name* of an *Identifier* is determined by the name resolution given below. Except for an <operation name> or <quoted operation name>, each distinct <name> in a particular specification always corresponds to a distinct *Token* and each occurrence of the <name> corresponds to the same *Token*. Similarly, each distinct <integer name>, <real name> or <string name> corresponds to a distinct *Token*, and each occurrence of this item corresponds to the same *Token*. In the case of an <operation name> or <quoted operation name>, the *Token* depends on the signature of the operation (the name, parameters and the result).

<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:

package	<package diagram>
system type	<system type diagram>
system	<typebased system definition> (and system diagram in [ITU-T Z.103]).
block type	<block type diagram>
block	<typebased block definition> (and block diagram in [ITU-T Z.103]).
process type	<process type diagram>
process	<typebased process definition> (and process diagram in [ITU-T Z.103]).
state type	<composite state type diagram>
state	<typebased composite state> (and typebased state partition definition or composite state diagram in [ITU-T Z.103]).
procedure	<procedure diagram>
signal	<signal definition> (for specialization, see [ITU-T Z.102])
type	<data type definition>
operator	<operation diagram> (for an **operator**)
method	<operation diagram> (for a **method** in [ITU-T Z.104])
interface	<interface definition>

It is allowed to omit the optional <scope unit kind> in a <path item> if the <name> of the <path item> otherwise uniquely determines the scope unit.

If given, the <scope unit kind> determines the qualifier kind (*Agent-type-qualifier, Agent-qualifier, State-type-qualifier, State-qualifier, Data-type-qualifier, Procedure-qualifier, Interface-qualifier*) and the resolution of the <name> of the <path item> is for that kind of scope unit. If no <scope unit kind> is given, the resolution of the <name> of a <path item> shall be a <name> of a scope unit and determines the qualifier kind (*Agent-type-qualifier, Agent-qualifier, State-type-qualifier, State-qualifier, Data-type-qualifier, Procedure-qualifier, Interface-qualifier*).

NOTE 2 – There is no <scope unit kind> corresponding to the scope units defined by <task area>.

The corresponding abstract syntax for the <scope unit kind> denoted by **operator** (or **method**, see [ITU-T Z.104]) is the implicit procedure that corresponds to the operation in the abstract syntax.

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. These definitions include <gate definition>s, <agent formal parameters> and <formal variable parameters> contained in the scope unit.

Entities are 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 (only when applying [ITU-T Z.102]);
g) variables (including formal parameters), synonyms (only when applying [ITU-T Z.104]);
h) literals, operators, (and methods see [ITU-T Z.104]);
i) remote variables (only when applying [ITU-T Z.104]);
j) sorts;
k) state types;
l) exceptions.

Each entity has its defining context in the scope unit that defines it.

Either the <qualifier> refers to a supertype or the <qualifier> reflects the logical 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 *Identifier* of an entity is then represented by the qualifier, the name of the entity, and, only for entities of kind h), the signature (see [ITU-T Z.104]). Each entity of a kind shall have an *Identifier* different from any other entity of the same kind.

NOTE 3 – Consequently, no two definitions in the same scope unit and belonging to the same entity kind are allowed to have the same <name>, except operations defined in the same <data type definition> that differ in at least one argument <sort> or the result <sort> (see [ITU-T Z.104]).

NOTE 4 – Any gate names occurring in channel definitions, state names, connector names, macro formal parameter names and macro names have special visibility rules and qualification is not possible. Special visibility rules are explained in the appropriate clauses.

It is possible to reference an entity 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 (not in Basic SDL-2010) or an instantiation of a type and the entity is visible in the base type; and
   1) it is not protected from visibility by restricted visibility (see [ITU-T Z.104]); and
   2) data specialization renaming has not been applied (see [ITU-T Z.104]); and

3) it is not a formal context parameter that has already been bound to an actual context parameter (see Recommendation [ITU-T Z.102]); or

c) the scope unit has a <package use clause> which mentions a <package diagram> 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 diagram> that is the defining context for the entity either has the <package public> clause omitted or <name> of the entity is mentioned in the <package public> clause; or

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

e) the scope unit contains a <data type definition> that is the defining context of the entity (in particular this applies for literals, operation signatures, and the implicit procedures that provide the behaviour of operations) and it is not protected from visibility by restricted visibility (see [ITU-T Z.104]); or

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

The binding of a <name> of an identifier to a definition through resolution by container proceeds in the following steps, starting with the scope unit denoted by the partial <qualifier> or (if no <qualifier> is given) the scope in which the <name> occurs, and considering every entity kind valid for the context where the name occurs:

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

b) if the scope unit is a specialization (not in Basic SDL-2010) or an instantiation of a type, step a) is repeated recursively through each base type in turn until the <name> is bound to an entity or a type is reached that has no base type; otherwise

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

d) if the scope unit has an <interface definition> and a unique entity exists (with the same <name> and a valid entity kind) 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 with 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 with which it is associated and contains the entities visible from the package.

NOTE 5 – In the concrete syntax, it is not possible to define packages inside other scope units. The above rule is only for defining the visibility rules that apply for packages.

When the <name> part of an <identifier> denotes an entity of the entity kind h), the binding of the <name> to a definition shall be resolvable by context. Resolution by context is attempted after resolution by container; that is, if binding of a <name> through resolution by container is possible, that binding is used even if resolution by context could bind that <name> to an entity also. Consequently, resolution by context is only applied if no unique binding is found through resolution by container (typically for a literal or operation of a data type). The context for resolving a <name> is an <assignment> (if the <name> occurred in an <assignment>), a <decision area> (if the <name> occurred in the <question> or <answer>s of a <decision area>), or an <expression> that is not part of any other <expression> otherwise. Resolution by context proceeds as follows.

1) 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> and a valid entity type for the context taking renaming into account.
2) Construct the product of the sets of <identifier>s associated with each <name>.
3) Consider only those elements in the product that 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.
4) When polymorphism is present in <assignment> (for example, in the support of object-oriented data), the static sort of an <expression> is not always the same as the static sort of the <variable> on the left hand side of the assignment, and similarly for the implicit assignments in parameters. The number of such mismatches is counted for each element.
5) Compare the elements in pairs, dropping those with more mismatches.
6) If there is more than one remaining element, all non-unique <identifier>s shall represent the same operation signature; otherwise in the context it is not possible to bind the <name>s to a definition.

Semantics

Each entity has an *Identifier* that has a *Qualifier* as the defining context and a *Name* that distinguishes this from other entities of the same kind in this context. When an entity is composite (such as an instance set or a structure) there are additional mechanisms to identify the components. For some entities (such as instances of signals or procedures) the *Name* is implicit and anonymous.

Model

It is allowed to omit some of the leftmost <path item>s, or the whole <qualifier> of an <identifier> if it is possible to uniquely expand the remaining <path item>s to a full <qualifier>. For example, if the <name> is unique to one entity in the whole specification, it is always allowed to omit the whole <qualifier>. If such a uniquely named entity is a scope unit, any <path item> to the left of this <name> in a <qualifier> is also allowed to be omitted.

When the <name or number> 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.

6.7 Empty clause

This clause is intentionally left blank.

6.8 Informal text

During the development of a specification, using informal text allows some parts to be informally specified for later formalization. If a specification contains informal text, it is not completely formally defined with a consequence that formally there is ambiguity.

Abstract grammar

Informal-text :: ...

Concrete 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 the Specification and Description Language. The semantics of the *Informal-text* is allowed to be defined by some other means. If during an interpretation of a specification *Informal-text* is interpreted, the future behaviour of the specification is not formally well-defined.

6.9 Text symbol

The <text symbol> is used in all kinds of <diagram>. The content depends on the diagram.

Unlike other symbols, text contained in a text symbol is considered as a single piece of text. For example, a <package text area> contains a single text piece that is a textual list of <signal definition list> and <data definition> items. By contrast, a <process symbol> for a <typebased process definition> contains a textual <typebased process heading> and separate textual pieces for each <gate>.

Concrete grammar

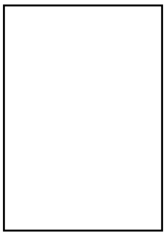
<text symbol> ::=

6.10 Frame symbol and page numbers

The frame symbol is used to frame pages of diagrams. The diagrams in Basic SDL-2010 are <package diagram>, <agent type diagram>, <procedure diagram>, <operation diagram> and <composite state type diagram>.

Concrete grammar

<frame symbol> ::=



When a <package use area> is associated with a <frame symbol>, the <package use area> shall be placed on the top of the <frame symbol>, for example, in a <block type page> or <process type page>.

A <frame symbol> contains a heading (such as <system type heading>, <process type heading>, <procedure heading>), which is a textual syntax non-terminal with a name ending in "heading". The heading is considered to be enclosed in an <implicit text symbol> that **contains** the heading. The heading (such as <system type heading>) is placed at the upper left corner of the <frame symbol>. The <implicit text symbol> is not visible (it is considered to be the same colour as the background) but implied and provides a clear separation between heading and any other text such as the <page number area> in the <frame symbol>. The <page number area> is also considered as contained in an <implicit text symbol>. The heading text is wrapped if necessary within the <implicit text symbol>.

<page number area> ::=
    [<page number> [**(** <number of pages> **)**]]

<page number> ::=
    <name or number>

<number of pages> ::=
<integer name>

The <page number area> is placed at the upper right corner of the <frame symbol>. It is a form of annotation and has no meaning in the abstract grammar.

The <page number> enables a name or number to be provided for a page of a diagram. The <number of pages> is optionally used to show how many pages there are in a diagram, which in Basic SDL-2010 would always be 1. In Comprehensive SDL-2010 a diagram is allowed to have extra pages.

7 Organization of Specification and Description Language specifications

It is not usually possible to describe a system in a single diagram. The language therefore supports the partitioning of the specification into a number of diagrams and use of packages in the language from elsewhere. [ITU-T Z.105] defines how an ASN.1 module is allowed as a package.

7.1 Framework

An <sdl specification> is described as a <system specification> (possibly augmented by a collection of <package diagram>s) or as a collection of <package diagram>s, in either case with <referenced definition>s. A <package diagram> allows definitions to be used in different contexts by "using" the package in these contexts (that is, in systems or packages which are otherwise independent). A <referenced definition> is a definition that is referenced from its defining context. Each <referenced definition> is logically "inserted" into exactly one place (the defining context) using a reference.

Abstract grammar

Sdl-specification :: [*Agent-definition*]
*Package-definition-**set***

The *Agent-definition* (if present) shall have an *Agent-type-identifier* for an *Agent-type-definition* with the *Agent-kind* **SYSTEM**.

Concrete grammar

<sdl specification> ::=
{
{ <package diagram> | <system specification> }
<referenced definition>* }***set***

<system specification> ::=
<typebased agent definition>[***is associated with*** <package use area>]

A <referenced definition> that is a <package diagram> represents a member of the *Package-definition-**set*** if the <qualifier> after the keyword **package** in the heading is omitted. Otherwise the <package diagram> is referenced from the context given by the <qualifier> and the <package diagram> represents a *Package-definition* in this context.

NOTE – How an <sdl specification> is stored on computer systems for analysis is not defined within this Recommendation or [ITU-T Z.100], or [ITU-T Z.102] to [ITU-T Z.106], and different approaches are allowed to be used. It is possible to store the whole <sdl specification> within a single computer file, or to have a file for each diagram or a file for each page of a diagram. Other schemes are also allowed.

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.

For an *Sdl-specification* with an *Agent-definition*, a type is "potentially instantiated" if it is either instantiated in the *Agent-definition*, or instantiated in a potentially instantiated type.

Model

The *Package-definition-**set*** of an *Sdl-specification* always includes the *Package-definition* of **package** `Predefined` defined in [ITU-T Z.104]; consequently this package does not have to be explicitly included as a <referenced 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, signals and interfaces.

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***
*Agent-type-definition-**set***
*Composite-state-type-definition-**set***
*Procedure-definition-**set***

For each *Agent-type-definition* there is a *Data-type-definition* that is an *Interface-definition* that has a *Sort* with the same *Name* as the *Agent-type-definition* (see clause 12.1.2).

Concrete grammar

<package diagram> ::=
<package page>

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

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

NOTE 1 – The <package name> has to be a <name>, whereas in SDL-2000 it is a name or a number (an <integer name> or a <real name>).

The <package heading> is placed in the top right-hand corner of the <frame symbol>.

The <package name> of the <package heading> represents the *Package-name* of the *Package-definition*.

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

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

Each <signal definition> of a <signal definition list> of a <package text area> represents a member of the *Signal-definition-**set*** of the *Package-definition*. Each <data definition> of a <package text area> represents a member of the *Data-type-definition* set of the *Package-definition* if it is a <data type definition> or <interface definition>, and a member of the *Syntype-definition* set of the *Package-definition* if it is a <syntype definition>.

<diagram in package> ::=
<package reference area>
| <entity in agent diagram>

<package reference area> ::=
<package symbol> ***contains*** <package name>

Each <package reference area> that is a <diagram in package> represents a member of the *Package-definition* set of the *Package-definition*. Each <entity in agent diagram> that is a <diagram in package> represents a member of the *Agent-type-definition* set of the *Package-definition* if it is an <agent type reference area>, a member of the *Composite-state-type-definition* set of the *Package-definition* if it is an <composite state type reference area>, and a member of the *Procedure-definition* set of the *Package-definition* if it is an <procedure reference area>.

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

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

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

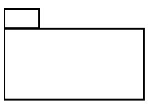
<selected entity kind> ::=
system type
| **block type**
| **process type**
| **package**
| **signal**
| **procedure**
| **type**
| **state type**
| **synonym**
| **signallist**
| **interface**

The keyword **signallist** has the same meaning as **interface** as a <selected entity kind>.

The keyword **synonym** is used only if [ITU-T Z.104] is being applied.

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

<package symbol> ::=



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

For each <package identifier> mentioned in a <package use clause>, there shall exist a corresponding <package diagram>. This package shall be part of <sdl specification> or a package contained in another package or else there shall exist a mechanism (not defined by the SDL-2010 Recommendations) for accessing the referenced <package diagram>, just as if it were a part of the <sdl specification>.

There shall be a <qualifier> in <package identifier> only if the package is logically contained in another package. If the corresponding <package diagram> is logically contained in another package, the <package identifier> reflects the hierarchical structure from the outermost <package diagram> to the defined <package diagram>.

The <package identifier> shall denote a visible package. All <package diagram>s in the <qualifier> of the fully qualified <package identifier> shall 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 for <identifier>. The visibility rules imply that a <package identifier> is made visible with a <package use clause> and that a package is visible in the scope in which it is logically contained. This scope extends also to the <package use clause> of the logical container package.

Likewise, if the <system specification> is omitted in an <sdl specification>, there shall exist a mechanism for using the <package diagram>s in other <sdl specification>s. The mechanism is not otherwise defined in this Recommendation.

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

The visibility of the name of an entity defined within a <package diagram> is explained in clause 6.6.

Signals that are not made visible in a **use** clause are visible if they are part of an interface made visible in a **use** clause, and therefore these signals 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>) shall be distinct within a <definition selection list>. For a <definition selection> in a <package public> clause, the <selected entity kind> is allowed to be omitted only if the name is used for just one defining occurrence directly in the <package diagram>. For a <definition selection> in a <package use clause>, <selected entity kind> is allowed to 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.

Semantics

A package enables a collection of types, signals and interfaces to be defined, so that it is possible to use them in a number of different systems or types. A package is allowed to contain other packages.

NOTE – The <system type diagram> for a system has to be logically enclosed in a package. The **package** Predefined is visible to the package enclosing the <system type diagram> because the package is included in the *Package-definition-**set*** of any *Sdl-specification*.

Model

If a package is mentioned in several <package use clause> items of a definition (in the same text area or different text areas of the definition), these are replaced by one <package use clause> in one text area that selects the union of the definitions selected in the <package use clause> items.

7.3 Referenced definition

Concrete grammar

<referenced definition> ::=
<diagram>

<diagram> ::=

<package diagram>
| <agent type diagram>
| <composite state type diagram>
| <procedure diagram>
| <operation diagram>

For each <referenced definition> except any outermost <package diagram>, there shall be a reference in the associated <package diagram> or <system specification>.

An optional <qualifier> and <name> is present in a <referenced definition> after the initial keyword(s). For each reference there shall 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> items of the same entity kind have the same <name>, the <qualifier> of one shall not constitute the leftmost part of the other <qualifier>, and neither <qualifier> is allowed to be omitted. The <qualifier> in a <referenced definition> shall be present if the <referenced definition> is a <package diagram> referenced from another context, except if the <package diagram> represents an outermost *Package-definition*.

The referenced definition is logically placed at the point of the reference to determine the properties of the system specification. That is, the abstract grammar for the referenced definition is determined by the context of the reference, which is the logical context for the referenced definition. In the case of a <package diagram> without a <qualifier> the *Package-definition* is a member of the *Package-definition-**set*** of the *Sdl-specification*.

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

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.

The language mechanisms introduced provide:

a) (pure) type definitions that are allowed to be defined anywhere in a system or in a package;

b) typebased instance definitions that define instances or instance sets according to types.

8.1 Types, instances and gates

There is a distinction between definition of instances (or set of instances) and definition of types. This clause introduces (in clause 8.1.1) type definitions for agents and composite states, while the introduction of other types are in procedures (clause 9.4), signals (clause 10.3), timers (clause 11.15), sorts (clause 12.1) and interfaces (clause 12.1.2). An agent type definition is not connected (by channels) to any instances; instead, agent type definitions introduce gates (clause 8.1.4). These are connection points on the typebased instances for channels.

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

An instance (or instance set) always has a 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*** *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*
Agent-kind	=	**SYSTEM** | **BLOCK** | **PROCESS**
Agent-type-identifier	=	*Identifier*
Agent-formal-parameter	=	*Parameter*
Parameter	::	*Variable-name* *Sort-reference-identifier* *Parameter-aggregation*
Parameter-aggregation	::	*Aggregation-kind*
State-machine	::	*State-name* *Nextstate-parameters* *Composite-state-type-identifier*

A system type definition (an *Agent-type-definition* with *Agent-kind* **SYSTEM**) shall not be logically contained in any other *Agent-type-definition*.

The *Agent-formal-parameter* list of a system shall be empty.

A process type definition (an *Agent-type-definition* with *Agent-kind* **PROCESS**) shall not contain a block type definition (an *Agent-type-definition* with *Agent-kind* **BLOCK**) or a block definition (an *Agent-definition* with *Agent-kind* **BLOCK**).

An *Agent-type-definition* shall have a *Name* that is different from the *Name* of every explicitly defined interface *Data-type-definition* or *Agent-definition* in the same scope.

NOTE 1 – This constraint on names is because every agent type has an implicitly defined interface with the same name, so the agent type has to 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 optional *Agent-type-identifier* of *Agent-type-definition* identifies the base type (super type) of a specialization.

NOTE 2 – Specialization is defined in [ITU-T Z.102] and is not included in Basic SDL-2010, and for Basic SDL-2010 *Agent-type-identifier* is always omitted, but the abstract syntax is included here so *Agent-type-definition* does not have to be redefined in [ITU-T Z.102].

For each member of an *Agent-type-definition-**set*** of the *Agent-type-definition* there is a corresponding *Data-type-definition* that is an *Interface-definition* that has a *Sort* with the same *Name* as the member of the *Agent-type-definition-**set*** (see clause 12.1.2).

For each member of an *Agent-definition-**set*** of the *Agent-type-definition* there is a corresponding *Data-type-definition* that is an *Interface-definition* that has a *Sort* with the same *Name* as the member of the *Agent-definition-**set*** (see clause 12.1.2).

For the *State-machine* of the *Agent-type-definition* there is a corresponding *Data-type-definition* that is an *Interface-definition* that has a *Sort* with the same *Name* as *State-name* of the *State-machine* (see clause 12.1.2).

The *State-machine* of the *Agent-type-definition* shall identify a *Composite-state-type-definition* that has a non-empty *Gate-identifier-**set*** that defines the gates of the state machine.

Concrete grammar

<agent type diagram> ::=
{ <system type diagram> | <block type diagram> | <process type diagram> }

<type preamble> ::=
{}

NOTE 3 – In Basic SDL-2010 <type preamble> is empty, but is added here to avoid the need to redefine agent type headings in [ITU-T Z.102].

<agent type additional heading> ::=
<agent additional heading>

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

<agent formal parameters> ::=
[<end>] **fpar** <aggregation kind> <parameters of sort>
{, <aggregation kind> <parameters of sort>}*

NOTE 4 – The syntax of SDL-2000 that uses round brackets rather than **fpar** is in [ITU-T Z.103].

The <agent formal parameters> of an <agent additional heading> represents the *Agent-formal-parameter* list of the *Agent-type-definition*, but the <agent formal parameters> of a <composite state type heading> represent a *Composite-state-formal-parameter* list of a *Composite-state-type-definition*.

<parameter aggregation> ::=
<aggregation kind>

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

NOTE 5 – The <<u>variable</u> name> has to be a <name>, whereas in SDL-2000 it is a name or a number (an <integer name> or a <real name>).

Each <<u>variable</u> name> of <parameters of sort> of <agent formal parameters> of <agent additional heading> of <agent type additional heading> represents a different *Agent-formal-parameter* with the <<u>variable</u> name> representing the *Variable-name* of the *Parameter* that is the *Agent-formal-parameter*. The <sort> of the <parameters of sort> represents the *Sort-reference-identifier* of each *Parameter* for the <parameters of sort>.

NOTE 6 – Agent parameters do not have a <parameter kind> and are always **in** parameters.

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.

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

The complete output set of an agent type is the union of all signals mentioned, either directly or as part of interfaces, in the outgoing signal lists associated with the gates of the agent type.

Other 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 formal parameters> list item with a <parameters of sort> that defines multiple parameter names is replaced by a sequence of <agent formal parameters> list items with the same <aggregation kind> each <parameters of sort> defining one name.

8.1.1.2 System type

A system type definition is a top-level agent type definition. It is denoted by the keywords **system type**.

Concrete grammar

<system type diagram> ::=
<system type page>

<system type page> ::=
<frame symbol>
contains {<system type heading> <page number area> <agent structure area> }
{ ***is connected to*** <gate on diagram> }*
[***is associated with*** <package use area>]

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

NOTE – The <system type name> has to be a <name>, whereas in SDL-2000 it is a name or a number (an <integer name> or a <real name>).

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

The <gate on diagram>s of a <system type diagram> shall be outside the diagram frame.

A <system type diagram> defines an *Agent-type-definition* with *Agent-kind* **SYSTEM**.

Each <gate on diagram> of the <system type page> represents a *Gate-definition-**set*** item of the *Agent-type-definition*.

8.1.1.3 Block type

Concrete grammar

<block type diagram> ::=
<block type page>

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

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

NOTE – The <block type name> has to be a <name>, whereas in SDL-2000 it is a name or a number (an <integer name> or a <real name>).

The <gate on diagram>s of a <block type diagram> shall be outside the diagram frame.

A <block type diagram> defines an *Agent-type-definition* with *Agent-kind* **BLOCK**.

Each <gate on diagram> of the <block type page> represents a *Gate-definition-**set*** item of the *Agent-type-definition*.

8.1.1.4 Process type

Concrete grammar

<process type diagram> ::=
<process type page>

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

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

NOTE – The <process type name> has to be a <name>, whereas in SDL-2000 it is a name or a number (an <integer name> or a <real name>).

The <gate on diagram>s of a <process type diagram> shall be outside the diagram frame.

A <process type diagram> defines an *Agent-type-definition* with *Agent-kind* **PROCESS**.

Each <gate on diagram> of the <process type page> represents a *Gate-definition-**set*** item of the *Agent-type-definition*.

8.1.1.5 Composite state type

Abstract grammar

Composite-state-type-definition	::	*State-type-name* *Composite-state-formal-parameter** *Gate-definition-**set*** *Data-type-definition-**set*** *Syntype-definition-**set*** *Composite-state-type-definition-**set*** *Variable-definition-**set*** *Procedure-definition-**set*** *Composite-state-graph*
Composite-state-formal-parameter	=	*Agent-formal-parameter*
Composite-state-type-identifier	=	*Identifier*

The *Gate-definition-***set** of a *Composite-state-type-definition* shall not be empty if there is a *State-machine* based on the *Composite-state-type-definition*.

Concrete grammar

<composite state type diagram> ::=
<composite state type page>

<composite state type page> ::=
<frame symbol>
contains {
<composite state type heading> <page number area>
<composite state structure area> }
{ ***is connected to*** <gate on diagram> }*
[***is associated with*** <package use area>]

Each <gate on diagram> of the <composite state type page> represents a *Gate-definition-**set*** item of *Composite-state-type-definition*.

<composite state type heading> ::=
<type preamble> **state type** [<qualifier>] <composite state type name>
[<agent formal parameters>]

NOTE 1 – The <composite state type name> has to be a <name>, whereas in SDL-2000 it is a name or a number (an <integer name> or a <real name>).

The <agent formal parameters> represent the *Composite-state-formal-parameter* list of the *Composite-state-type-definition*.

Each <variable name> of <parameters of sort> of <agent formal parameters> of <composite state type heading> represents a different *Composite-state-formal-parameter* with the <variable name> representing the *Variable-name* of the *Parameter* that is the *Agent-formal-parameter* that is the *Composite-state-formal-parameter*. The <sort> of the <parameters of sort> represents the *Sort-reference-identifier* of each *Parameter* for the <parameters of sort>.

NOTE 2 – Composite state parameters do not have a <parameter kind> and are always **in** parameters.

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

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

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 clause 11.11.

8.1.2 Type expression

A type expression is used to define the properties of an instance (or set of instances) in terms of a type. In Basic SDL-2010 a type expression simply identifies a type.

Concrete grammar

<type expression> ::=
<base type>

<base type> ::=
<identifier>

A <type expression> yields the type identified by the identifier of <base type>.

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

NOTE – The static properties on the usage of a <type expression> are violated if an output in a scope unit refers to a gate or a channel that is not defined for the nearest enclosing type having gates, or if there is no communication path to the gate. Instantiation of that type results in an erroneous specification.

8.1.3 Empty clause

This clause is intentionally left blank.

8.1.4 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 with other instances or with gates on 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*

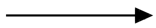
Out-signal-identifier = *Signal-identifier*

Concrete grammar

<gate on diagram> ::=
    <gate definition>

<gate definition> ::=
    <gate symbol 1>
    is associated with { <gate> <signal list area> }***set***
  | <gate symbol 2>
    is associated with { <gate> <signal list area> <signal list area> }***set***

<gate symbol 1> ::=



<gate symbol 2> ::=

<gate> ::=
    <gate name>

NOTE 1 – The <gate name> has to be a <name>, whereas in SDL-2000 it is a name or a number (an <integer name> or a <real name>).

NOTE 2 – If [ITU-T Z.103] is applied, it is permitted to omit the <signal list area> if it is possible to derive the signal list from other communication path information.

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

The <signal list area> elements are associated with the directions of the <gate symbol 1> or <gate symbol 2> as denoted by the arrowheads. A <signal list area> shall be unambiguously close enough to the arrowhead to which it is associated. The arrowhead indicates whether the <signal list area> represents an *In-signal-identifier****-set*** or an *Out-signal-identifier****-set***. An *In-signal-identifier* is represented by a <signal list item> element of the <signal list area> associated with an arrowhead at the end of a <gate symbol 1> or <gate symbol 2> connected to the diagram frame. An *Out-signal-identifier* is represented by a <signal list item> element of the <signal list area> associated with an arrowhead at the end of a <gate symbol 1> or <gate symbol 2> not connected to the diagram frame.

If the type denoted by <base type> in a <typebased block definition> or <typebased process definition> contains channels, the following rule applies: for each combination of a gate, a signal, and the direction of the <signal list> of the gate defined by the type, the type shall contain at least one channel that – for the given direction – is connected to the frame at this gate and mentions the signal (or has no explicit <signal list area> associated if [ITU-T Z.103] is being applied to allow the <signal list area> to be omitted).

Semantics

The use of gates in type definitions corresponds to the use of communication paths in the enclosing scope in (a set of) instance specifications.

8.2 Type references and operation references

Type diagrams have type references. The referenced diagram defines the properties of the type. The type is fully described in the referenced diagram.

An operation reference is a special case of a type reference, because there is no symbol for an operation reference and instead the reference is given textually. The use of operation reference is allowed only within a data type definition, which is only textual, so a symbol could not be used to denote the reference.

The same type definition is allowed to have several type references. If there are several references to the same type in a scope unit, this is the same as having one reference.

To enable the concrete grammar to be extended more easily in [ITU-T Z.102], [ITU-T Z.103] and [ITU-T Z.104], the concrete syntax given for type references has some use of rules (rather than syntax) to limit allowed productions, and several non-terminal productions are introduced here.

Concrete grammar

Each of the following is a type reference: <agent type reference area>, <composite state type reference area>, <procedure reference area>.

In Basic SDL-2010, identity of the scope unit directly enclosing the type reference is used with the <name> given in the type reference to determine the *Identifier* of the referenced type: the type reference and the referenced type are logically in the same scope unit – the scope unit containing the reference.

There shall be at least one type reference for the <referenced definition> in the logically containing scope unit. This enables the <referenced definition> to be located, so that it is possible to map the concrete diagrams to the logically enclosing scope in the complete system specification in the abstract grammar.

<agent type reference area> ::=
{ <system type reference area>
| <block type reference area>
| <process type reference area> }

<system type reference area> ::=
<system type symbol> ***contains*** { **system** <system type name> }

NOTE 1 – The keyword **system** in a <system type reference area> for a system allows a type based system to be more easily distinguished from a type based block set. The keyword **system** was not allowed in SDL-2000, but was required by some tools.

<block type reference area> ::=
<block type symbol> ***contains*** <block type name>

<process type reference area> ::=
<process type symbol> ***contains*** <process type name>

<composite state type reference area> ::=
<composite state type symbol> ***contains*** <composite state type name>

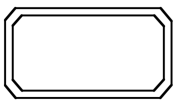
<procedure reference area> ::=
<procedure symbol> ***contains*** <procedure reference heading>

<procedure reference heading> ::=
<procedure name>

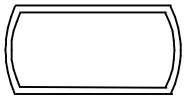
<system type symbol> ::=
<block type symbol>

<block type symbol> ::=

<process type symbol> ::=



<composite state type symbol> ::=



<procedure symbol> ::=

<operation reference> ::=
<operation kind> <operation signature> **referenced** <end>

<operation kind> ::=
{ **operator** }

<arguments> and <result> of the <operation signature> in an <operation reference> are allowed to be omitted if there is no other <operation reference> within the same sort of data that has the same name. In this case, the referenced <operation diagram> is identified simply by its name. The <operation reference> enables the referenced <operation diagram> to be located, so that it is possible to map the concrete diagrams to the enclosing data type definition in the logical hierarchy in the abstract grammar.

9 Agents

An agent definition defines an (arbitrarily large) set of agents. An agent is characterized by having variables, procedures, a state machine (based on a 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: only one at a given time.

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.

Abstract grammar

Agent-definition	::	*Agent-name* *Number-of-instances* *Agent-type-identifier*
Number-of-instances	::	*Initial-number [Maximum-number] Lower-bound*
Initial-number	=	*Nat*
Maximum-number	=	*Nat*
Lower-bound	=	*Nat*

If there is a *Maximum-number*, the *Initial-number* of instances shall be less than or equal to *Maximum-number* and *Maximum-number* shall be greater than zero. The *Lower-bound* shall be less than or equal to the *Initial-number*.

Concrete grammar

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

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

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

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

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

<lower bound> ::=
<Natural simple expression>

<agent structure area> ::=
{ {<agent text area>}*
{<entity in agent diagram>}*
<interaction area> }***set***

<agent text area> ::=
<text symbol>
contains {
{ <valid input signal set>
| <signal definition list>
| <variable definition>
| <data definition>
| <timer definition> }* }

An <agent text area> of a system type or block type shall not contain a <variable definition> in Basic SDL-2010. The state machine of a system or block is allowed to define local variables.

Each <data definition> of an <agent text area> represents a member of the *Data-type-definition* set of the *Agent-type-definition* for the agent if it is a <data type definition> or <interface definition>, and a member of the *Syntype-definition* set of the *Agent-type-definition* for the agent if it is a <syntype definition>.

<entity in agent diagram> ::=
<agent type reference area>
| <composite state type reference area>
| <procedure reference area>

<interaction area> ::=
{ <state machine area> { <agent area> | <channel definition area>}* }***set***

<state machine area> ::=
<state symbol> ***contains*** { <typebased composite state> { <gate>*}***set*** }

The <gate>s contained in a <state symbol> in a <state machine area> are placed near the border of the symbol and associated with the connection point to channels. They are placed close to the endpoint of the channels at the <state symbol>. Each <gate> shall have a <gate name> that identifies a *Gate-definition* of the *Composite-state-type-definition* identified by the *State-machine*.

The <state machine area> of <interaction area> defines the state machine (composite state) of the agent. In Basic SDL-2010, the state machine is defined by <state machine area> with a <typebased composite state>. The <state name> that is the <composite state name> of the <typebased composite state> represents the *State-name* of the *State-machine* of the *Agent-type-definition*. The <nextstate parameters> of the <typebased composite state> represent the *Nextstate-parameters* of the *State-machine* of the *Agent-type-definition*. The type of the <typebased composite state> represents the *Composite-state-type-identifier* of the *State-machine* of the *Agent-type-definition*.

<agent area> ::=
<typebased agent definition>

An <agent area> represents an *Agent-definition* as further described for <typebased system definition>, <typebased block definition> and <typebased process definition>.

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

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 (it is omitted in *Number-of-instances*). If the <lower bound> is omitted, the *Lower-bound* is zero.

The <valid input signal set> of an agent defines signals in the valid input signal set of the state machine of the agent. Signals occurring in an explicitly defined <valid input signal set> and not defined for a communication path allow communication between instances within the same instance set.

Semantics

An *Agent-definition* has a *Name*, which it is allowed to use as *a Path-item* of a *Qualifier* for items defined within the agent (system, block or process depending on the kind of the agent).

An *Agent-definition* defines a set of agent instances. It is possible for several agent instances in the set to 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 *Initial-number* 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 *Maximum-number* (if present) represents the maximum number of simultaneous instances of the agent set. The last value *Lower-bound* represents the minimum number of simultaneous instances of the agent set that shall exist at any time after the initial instances have been created. If *Maximum-number* and *Lower-bound* have the same value, it is not possible to create or stop instances.

Some 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. The system agent is the special case of a block not contained within another block.

An agent instance of an *Agent-definition* has a communicating extended finite state machine defined by the *State-machine* of the *Agent-type-definition* identified by the *Agent-type-identifier* of the *Agent-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. It is possible that the time taken to create an agent is significant or insignificant. 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 are created by create request actions of agents being interpreted; their interpretations start when their start action is interpreted and they cease to exist by performing stop actions.

When the state machine of an agent interprets a stop, it enters a "stopping condition". The state machine of such an agent remains in the 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, if any, introduced for each global variable as described in [ITU-T Z.102], [ITU-T Z.103] and [ITU-T Z.104]). After an agent has terminated, its pid is no longer valid.

The way the state machine of an agent behaves is determined by the *Composite-state-type-definition* identified by the *Composite-state-type-identifier* of the *State-machine* of the *Agent-type-definition* identified by the *Agent-type-identifier* of the *Agent-definition*. Contained agents and variables (including agent parameters) initialized before the *Nextstate-parameters* of the *State-machine* are evaluated and the composite state is invoked in the same way as entering a composite state from a *Nextstate-node*. If the composite state interprets a return to the agent, this is interpreted as a *Stop-node* and the agent enters the stopping condition. The simplest *Composite-state-type-definition* has a *Composite-state-graph* that has a *State-start-node* with a *Transition* that has an empty *Graph-node* list and that has a *Return-node* as a *Terminator*. For an agent with such a minimal state machine, as soon as all the initial contained agents have been created the agent enters a stopping condition. An agent with no contained initial instances and only a minimal state machine therefore ceases to exist as soon as it is created.

Signals received by agent instances are denoted as input signals, and signals sent from agent instances are denoted as output signals.

NOTE 1 – Calling and serving remote procedure calls, and accessing remote variables, also correspond to exchange of signals (see [ITU-T Z.102], [ITU-T Z.103] and [ITU-T Z.104]), but these features are not parts of Basic SDL-2010.

Signals are 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 state machine of the agent;

b) the valid input signal set defined explicitly for the agent;

c) the valid input signal set defined explicitly for the state machine of the agent;

d) the implicit input signals introduced (see [ITU-T Z.102], [ITU-T Z.103] and [ITU-T Z.104]); and

e) 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 of the agent are delivered to this input port of the agent, provided that the signal appears on a channel connected to its state machine. If a signal conveys an availability time, the signal is not delivered to the input port until this time has been reached.

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 clause 11.7). When waiting in a state, an input signal whose identifier is not in the save signal set is taken from the input port and consumed by the agent and the associated transition is initiated.

The input port is able to retain any number of input signals, so that it is possible that several input signals are queued for the finite state machine of the agent instance. For each signal the identity of the gate (of the *Composite-state-type-definition* of the *State-machine* of the local agent) via which it arrived is retained, so that this is available to decide which transition to initiate when the signal is consumed. The set of retained signals is ordered in the queue for delivery according to their availability time, which for each signal that does not convey an availability time is the same as its arrival time (that is, the signal is available as soon as it arrives). Two or more signals are arbitrarily ordered, if they have the same availability time and same signal priority. For signals that do not convey an availability time and arrive on different paths it is possible that the sequence of the arrival events is not determined (they are "simultaneous") and therefore the signals have the same availability time. For a signal that does not convey an availability time it is also possible that the arrival time (and therefore availability time) of the signal has the same availability time of a signal that previously arrived with an availability time. It is also possible that two signals with availability time have the same availability time. Signal instances that are "simultaneous" and convey different signal priorities are ordered according to the signal priority value, so that signals with lower values are before signals with higher values.

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 body>, **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.

The definition of an agent implies the definition of an interface in the same scope of the agent (see clause 12.1.2). The pid sort 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 is required to 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 name clashes.

The complete output set of an agent set is the same as the complete output set of the type of the agent set.

In all agent instances, there are four expressions: *Self-expression*, *Parent-expression*, *Offspring-expression* and *Sender-expression*. They give a result for:

a) the agent instance (*Self-expression*) of the pid sort of the agent;

b) the creating agent instance (*Parent-expression*) of the `Pid` sort;

c) the most recent agent instance created by the agent instance (*Offspring-expression*) of the `Pid` sort;

d) the agent instance from which the last input signal has been consumed (*Sender-expression*) (see also (a)) of the `Pid` sort.

Each *Pid-expression* above gives the value of an anonymous variable of the agent (referred to here as **self**, **parent**, **offspring**, and **sender**, respectively, and further explained in clause 12.3.4.2).

For all agent instances created when the containing instance is created, **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**. The semantics of agents applies with the additions provided in this subclause.

Abstract grammar

An agent with the *Agent-kind* **SYSTEM** has *Agent-definition* with an *Agent-type-identifier* that identifies an *Agent-type-definition* with an *Agent-kind* **SYSTEM**. An agent with the *Agent-kind* **SYSTEM** shall not be contained in any other agent.

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.

Concrete grammar

<typebased system definition> ::=
    <block symbol> ***contains*** <typebased system heading>

<typebased system heading> ::=
    system <<u>system</u> name> <colon> <<u>system</u> type expression>

NOTE 1 – The <<u>system</u> name> has to be a <name>, whereas in SDL-2000 it is a name or a number (an <integer name> or a <real name>).

NOTE 2 – <number of instances> is not allowed in a <typebased system heading>, because the number is always 1.

A <typebased system definition> defines an *Agent-definition* with *Agent-kind* **SYSTEM** that is an instantiation of the system type denoted by the <system type expression>. The <system name> represents the *Agent-name* of the *Agent-definition*. The <base type> of the <system type expression> represents the *Agent-type-definition* of the *Agent-definition*.

Semantics

An *Agent-definition* with the *Agent-kind* **SYSTEM** is the Specification and Description Language 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 takes place using signals. Within a system, the communication signals are conveyed on 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 identified by an *Agent-definition* with the *Agent-kind* **SYSTEM**. The interpretation of a system instance is performed by an abstract Specification and Description Language machine, which thereby gives semantics to the Specification and Description Language 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 state machine of the system.

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.

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 or the system. All communication between different contained instances directly within the block is performed asynchronously using signal exchange.

Concrete grammar

<typebased block definition> ::=
<block symbol> ***contains*** { <typebased block heading> { <gate>* }***set*** }

<block symbol> ::=

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

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

NOTE – The <block name> has to be a <name>, whereas in SDL-2000 it is a name or a number (an <integer name> or a <real name>).

A <typebased block definition> defines an *Agent-definition* of *Agent-kind* **BLOCK** that is an instantiation of the block type denoted by the <block type expression>. The <block name> represents the *Agent-name* of the *Agent-definition*. The <base type> of the <block type expression> represents the *Agent-type-definition* of the *Agent-definition*.

Semantics

A block definition is an agent definition that defines a container for a state machine (possibly with minimal behaviour) and zero or more process or block definitions.

A block instance is an instantiation of a block type identified 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.

In a block the state machine of the block 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.

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 used to introduce shared data into a specification, allowing the variables of the containing process to be used. All instances in a process are able to 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 a signal, it is in a state; therefore it is possible for an alternate instance to be interpreted.

Abstract grammar

Any contained *Agent-definition* of an *Agent-definition* with the *Agent-kind* **PROCESS** shall have the *Agent-kind* **PROCESS**.

Concrete grammar

<typebased process definition> ::=
<process symbol> ***contains*** { <typebased process heading> { <gate>* }***set*** }

<process symbol> ::=

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

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

NOTE – The <process name> has to be a <name>, whereas in SDL-2000 it is a name or a number (an <integer name> or a <real name>).

A <typebased process definition> defines an *Agent-definition* with *Agent-kind* **PROCESS** that is an instantiation of the process type denoted by the <process type expression>. The <process name> represents the *Agent-name* of the *Agent-definition*. The <base type> of the <process type expression> represents the *Agent-type-definition* of the *Agent-definition*.

Semantics

A process definition is an agent definition that defines a container for a state machine (possibly with minimal behaviour) and zero or more process definitions. A process instance is an instantiation of a process type identified by an *Agent-definition* with the *Agent-kind* **PROCESS**.

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. Alternating interpretation implies that only one of the instances inside the alternating context interprets a transition at a time, and also that once interpretation of a transition of an involved process instance has started, it continues until a state is reached or the process instance terminates.

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 are 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*

The optional *Procedure-identifier* of *Procedure-definition* identifies the base type (super type) of a specialization.

NOTE 1 – Specialization is defined in [ITU-T Z.102] and is not included in Basic SDL-2010, and for Basic SDL-2010 *Procedure-identifier* is always omitted, but the syntax is included here so *Procedure-definition* does not have to be redefined in [ITU-T Z.102].

If a *Procedure-definition* contains *Result*, it corresponds to a value returning procedure.

Procedure-name	=	*Name*
Procedure-formal-parameter	= | |	*In-parameter* *Inout-parameter* *Out-parameter*
In-parameter	::	*Parameter*
Inout-parameter	::	*Parameter*
Out-parameter	::	*Parameter*
Result	::	*Sort-reference-identifier* *Result-aggregation*
Result-aggregation	::	*Aggregation-kind*
Procedure-graph	::	[*Procedure-start-node*] *State-node-**set*** *Free-action-**set***
Procedure-start-node	::	*Transition*
Procedure-identifier	=	*Identifier*

In an *Sdl-specification*, all potentially instantiated procedures shall have a *Procedure-start-node*.

If a *Procedure-definition* is identified by the *Procedure-identifier* of an *Operation-signature* it defines how an operation behaves. The *Procedure-graph* of the *Procedure-definition* for an operation shall not contain a *State-node* (explicit or implicit). The *Procedure-definition-**set*** of the *Procedure-definition* for an operation shall be empty.

Concrete grammar

<procedure diagram> ::=
<procedure page>

<procedure page> ::=
<frame symbol> ***contains*** {
<procedure heading> <page number area>
{ <procedure text area>*
<composite state type reference area>*
<procedure reference area>*
<procedure body area> }***set*** }
[***is associated with*** <package use area>]

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

<procedure heading> ::=
<procedure preamble>
procedure [<qualifier>] <procedure name>
[<procedure formal parameters>]
[<procedure result>]

NOTE 2 – The <procedure name> has to be a <name>, whereas in SDL-2000 it is a name or a number (an <integer name> or a <real name>).

<procedure preamble> ::=
<type preamble>

<procedure formal parameters> ::=
[<end>] **fpar** <formal variable parameters> {, <formal variable parameters> }*

NOTE 3 – The syntax of SDL-2000 that uses round brackets rather than **fpar** is in [ITU-T Z.103].

<formal variable parameters> ::=
`<parameter kind> <parameter aggregation> <parameters of sort>`

<parameter kind> ::=
in/out | in | out

Each <variable name> of <parameters of sort> of <formal variable parameters> of <procedure formal parameters> represents a different *Procedure-formal-parameter* with the variable name> representing the *Variable-name* of the *Parameter*, which is an *In-parameter* if <parameter kind> is **in**, and *Out-parameter* if <parameter kind> is **out**, and an *Inout-parameter* if <parameter kind> is **in/out**. The <parameter kind> is the one before the <parameters of sort>. The <sort> of the <parameters of sort> represents the *Sort-reference-identifier* of each *Parameter* for the <parameters of sort>. The <parameter aggregation> before the <parameters of sort> represents the *Parameter-aggregation* of each *Parameter* for the <parameters of sort>.

NOTE 4 – In Basic SDL-2010 the parameter kind has to be given. In [ITU-T Z.103] it is optional and the default is **in**.

<procedure result> ::=
returns <result aggregation> <sort>

NOTE 5 – In Basic SDL-2010 it is not allowed to name a variable for the result.

<result aggregation> ::=
<aggregation kind>

<entity in procedure> ::=
<variable definition>
| <data definition>

<procedure text area> ::=
<text symbol> ***contains***
{ <entity in procedure> }*

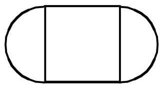
Each <variable definition> as an <entity in procedure> of a <procedure text area> represents a member of the *Variable-definition* set of the *Procedure-definition*.

Each <data definition> as an <entity in procedure> of a <procedure text area> represents a member of the *Data-type-definition* set of the *Procedure-definition* if it is a <data type definition> or <interface definition>, and a member of the *Syntype-definition* set of the *Procedure-definition* if it is a <syntype definition>.

<procedure body area> ::=
{ [<procedure start area>]
{<state area> | <in connector area> }* } ***set***

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

<procedure start 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.

A procedure variable is a local variable within the procedure. A procedure variable is not allowed to 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 area>; see clause 11.13.3), a *Value-returning-call-node* (represented by a <value returning procedure call>; see clause 12.3.5), or an *Operation-application* (represented by an <operation application>; see clause 12.2.6) causes the creation of a procedure instance as part of the agent or procedure instance being interpreted and the interpretation of the new procedure instance to commence in the following way.

a) A local variable is created for each *In-parameter*, having the *Name* and *Sort* and *Parameter-aggregation* 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* and *Parameter-aggregation* 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 clause 11.13.3. The contained *Variable-name* is used throughout the interpretation of the *Procedure-graph* when referring to the data item associated with 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*, each *Out-parameter* is given the data item 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.

NOTE 6 – The *Call-node* or *Value-returning-call-node* is always in the same enclosing agent as the *Procedure-definition*, because a subtype *Procedure-definition* is implicitly created locally if necessary (see clause 11.13.3).

Model

A <formal variable parameters> with a <parameters of sort> that defines multiple parameter names is replaced by a sequence of <formal variable parameters> with the same <parameter kind> and <aggregation kind>, and each <parameters of sort> defining one name.

10 Communication

10.1 Channel

Abstract grammar

Channel-definition	::	*Channel-name* [**NODELAY**] *Channel-path**-set***
Channel-path	::	*Channel-endpoint* *Originating-gate* *Channel-endpoint* *Destination-gate* *Signal-identifier**-set***
Originating-gate	=	*Gate-identifier*
Destination-gate	=	*Gate-identifier*
Gate-identifier	=	*Identifier*
Agent-identifier	=	*Identifier*
Channel-name	=	*Name*
Channel-endpoint	=	*Agent-identifier* *State-identifier* **ENV**
State-identifier	=	*Identifier*

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* shall be the same as the *Destination-gate* of the other *Channel-path*.

If the *Originating-gate* and the *Destination-gate* are gates of the same agent, the channel shall be unidirectional (there shall be only one element in the *Channel-path**-set***).

If the *Originating-gate* and the *Destination-gate* are both gates of the same state machine, the channel shall be unidirectional (there shall be only one element in the *Channel-path**-set***).

The *Originating-gate* or *Destination-gate* shall be defined in the same scope unit (which includes directly enclosed scopes) 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.

Each gate and the channel shall have at least one common element in their signal lists in the same direction.

Concrete grammar

<channel definition area> ::=
<channel symbol 1>
is associated with
{ <channel name> <signal list area> }***set***
is attached to {
{ <agent area> | <state machine area> | <gate on diagram> }
{ <agent area> | <state machine area> | <gate on diagram> } }***set***
| <channel symbol 2>
is associated with
{ <channel name> <signal list area> <signal list area> }***set***
is attached to {
{ <agent area> | <state machine area> | <gate on diagram> }
{ <agent area> | <state machine area> | <gate on diagram> } }***set***

NOTE – The <channel name> has to be a <name>, whereas in SDL-2000 it is a name or a number (an <integer name> or a <real name>). In Basic SDL-2010 the channel name has to be given.

If the <channel symbol 1> or <channel symbol 2> is attached to an <agent area> that is a <typebased agent definition>, there shall be a <gate> in the <typebased agent definition> placed near the channel attachment to the area. If the <channel symbol 1> or <channel symbol 2> is attached to a <state machine area> defined by a <typebased composite state>, there shall be a <gate> in the <state machine area> placed near the channel attachment to the area. This <gate> in the <agent area> or <state machine area> represents either the *Destination-gate* or *Originating-gate*, with the other gate determined by the other end of the channel. The <gate> shall be closer to the channel attachment to the area than any other <gate> in the area. The <gate> in the area attached to a channel identifies a *Gate-definition* of the agent or state machine. For an <agent area>, the <gate> identifies the *Gate-definition* of the *Agent-type-identifier* of the *Agent-definition* corresponding to the area, where the *Gate-definition* has the *Gate-name* given by <gate>. For a <state machine area> the <gate> identifies the *Gate-definition* of the *Composite-state-type-identifier* of the *State-machine* corresponding to the area, where the *Gate-definition* has the *Gate-name* given by <gate>.

For a <channel symbol 1>, there is only one *Channel-path* in the *Channel-definition*.

If the arrowhead of a <channel symbol 1> points away from an attached <agent area>, the first *Channel-endpoint* is the *Agent-identifier* for that agent. If the arrowhead of a <channel symbol 1> points away from an attached <state machine area>, the first *Channel-endpoint* is the *State-identifier* for the state machine. Otherwise, if the <channel symbol 1> points away from an attached <gate on diagram> the first *Channel-endpoint* is **ENV**.

If the arrowhead of a <channel symbol 1> points away from an attached <agent area> (or <state machine area>), the <gate> in this area represents the *Gate-identifier* of the *Originating-gate* of the *Channel-path* and identifies the *Gate-definition* of this agent (or state machine respectively). If the arrowhead of a <channel symbol 1> points away from an attached <gate on diagram>, this gate represents the *Originating-gate*.

If the arrowhead of a <channel symbol 1> points to an attached <agent area>, the second *Channel-endpoint* is the *Agent-identifier* for that agent. If the arrowhead of a <channel symbol 1> points to an attached <state machine area>, the second *Channel-endpoint* is the *State-identifier* for the state machine. Otherwise, if the <channel symbol 1> points to an attached <gate on diagram> the second *Channel-endpoint* is **ENV**.

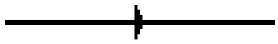
If the arrowhead of a <channel symbol 1> points to an attached <agent area> (or <state machine area>), the <gate> in this area represents the *Gate-identifier* of the *Destination-gate* of this *Channel-path* and identifies the *Gate-definition* of this agent (or state machine respectively). If the arrowhead of a <channel symbol 1> points to an attached <gate on diagram>, this gate represents the *Destination-gate*.

For a <channel symbol 2> there are two *Channel-path* items: one arrowhead corresponds to one *Channel-path* and the other arrowhead to the other *Channel-path*.

<channel symbol 1> ::=
<delaying channel symbol 1>
| <nondelaying channel symbol 1>

<channel symbol 2> ::=
<delaying channel symbol 2>
| <nondelaying channel symbol 2>

<delaying channel symbol 1> ::=



<delaying channel symbol 2> ::=

<nondelaying channel symbol 1> ::=

<nondelaying channel symbol 2> ::=

The symbols <delaying channel symbol 1>, <delaying channel symbol 2>, <nondelaying channel symbol 1> and <nondelaying channel symbol 2> are line symbols (see *Concrete grammar* in clause 5.3.2 of [ITU-T Z.100]).

For each arrowhead on the <channel symbol 2>, there shall be one <signal list area> close enough to the arrowhead to be associated unambiguously with the arrowhead (compared with any other <signal list area>) and represents the *Signal-identifier***-set** for the corresponding *Channel-path*. The arrowhead indicates the direction of the channel path for the signal list associated with it. The *Channel-endpoint* items, *Originating-gate* and *Destination-gate* for this *Channel-path* are determined in the same way as for the arrowhead on a <channel symbol 1>.

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 and represents **NODELAY** in the *Channel-definition*.

A channel with both endpoints being gates of one <typebased agent definition> represents individual channels from each of the agents in this set to all agents in the set, including the originating agent. In Basic SDL-2010 such channels shall be defined in one direction only using <channel symbol 1>.

NOTE – A bi-directional channel connecting an agent in the set to the agent itself is split into two unidirectional channels by a model discussed in [ITU-T Z.102].

In Basic SDL-2010 all channels are explicit. When [ITU-T Z.102] and [ITU-T Z.103] are applied, there are also implicit channels. The following rules ensure that implicit channels are not required (that is, the implicit channels in [ITU-T Z.102] or [ITU-T Z.103] are not needed):

One signal list element (interface, or signal) matches another signal list element if:

a) both denote the same interface or both denote the same signal; or

b) the first denotes a signal and the second denotes an interface and the interface includes the signal.

In the following rules an instance is the state machine of the agent type or an instance of an enclosed agent.

Rule 1: Ensuring there are no implicit channels between entities inside one agent type

a) If an element of the outgoing signal list associated with a gate of an instance in an agent type matches an element of an incoming signal list associated with a gate of another instance in the same agent type, then

b) at least one of these gates shall have an explicit channel attached to it.

Rule 2: Ensuring there are no implicit channels from the incoming gates on an agent type

a) If an element of the incoming signal list associated with a gate outside an agent type matches an element of an incoming signal list associated with a gate of an instance in the agent type, then

b) there shall be either an explicit channel inside the agent type attached to the gate outside the agent type, or an explicit channel attached to the gate of the instance inside the agent type.

Rule 3: Ensuring there are no implicit channels to the outgoing gates on instances

a) If an element of the outgoing signal list associated with a gate outside an agent type matches an element of an outgoing signal list associated with a gate of an instance in the agent type, then

b) there shall be either an explicit channel inside the agent type attached to the gate outside the agent type, or an explicit channel connected to the gate of the instance inside the agent type.

Semantics

A *Channel-definition* represents a transportation path for signals (including the implicit signals implied by remote procedures and remote variables, see [ITU-T Z.102], [ITU-T Z.103] and [ITU-T Z.104]). A channel is considered as one or two independent unidirectional channel paths between two agents or between an agent and its environment. Alternatively, a channel connects the state machine (composite state) of an agent with the environment or with a contained agent.

The *Signal-identifier*-**set** in each *Channel-path* in the *Channel-definition* contains the signals that are 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 their origin. If two or more signals are presented simultaneously to the channel, they are arbitrarily ordered.

A channel with delay is allowed to 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 possibly non-constant time interval, the first signal instance in the queue is released and given to one of the endpoints that is attached to the channel.

It is possible that several channels exist between the same two endpoints. It is possible to convey the same signal type 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.

10.2 Connection

A connection is the point where a channel inside a frame symbol for an agent diagram is connected to names of one or more channels outside a frame symbol. This feature is associated with agent diagrams, which are not included in Basic SDL-2010.

10.3 Signal

Abstract grammar

Signal-definition	::	*Signal-name* *Signal-parameter**
Signal-parameter	::	*Aggregation-kind* *Sort-reference-identifier*
Signal-identifier	=	*Identifier*
Signal-name	=	*Name*

The *Signal-parameter* list is a list of the aggregation kind and sort for each parameter defined for this signal type.

The *Identifier* of a *Signal-identifier* shall either identify a *Signal-definition* (a signal) or a *Timer-definition* (a timer signal).

Concrete grammar

<signal definition list> ::=

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

<signal definition> ::=

<type preamble>
<signal name>
[<sort list>]

NOTE – The <signal name> has to be a <name>, whereas in SDL-2000 it is a name or a number (an <integer name> or a <real name>).

<sort list> ::=

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

Each <signal definition> represents one *Signal-definition*. Each <aggregation kind> and <sort> in the <sort list> of a <signal definition> adds the *Aggregation-kind* and *Sort-reference-identifier* of a *Signal-parameter* to the end of the *Signal-parameter* list.

If several <signal definition> items are specified in one <signal definition list>, this is equivalent to individual <signal definition list>s for each of them.

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 is 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.

10.4 Signal list area

A signal list and signal list area are used to define the communication items (signals, interfaces, etc.) associated with a gate, channel or input.

Concrete grammar

<signal list area> ::=

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

<signal list symbol> ::=

<signal list> ::=

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

<signal list item> ::=

<signal identifier>
| <timer identifier>
| **(** <interface identifier> **)**

A <signal list item>, which is an <identifier>, denotes a <signal identifier> or <timer identifier>; otherwise the bracketed <identifier> shall be an <interface identifier>.

NOTE – The entity kind for signals is the same as the entity kind for timers (and interfaces), therefore it is not allowed to have the same name for a signal and a timer (or for a signal and an interface, or a timer and an interface) in the same scope, so a name of a <signal list item> always resolves to a unique item in a given scope.

A <signal list> of a <signal list area> denotes a *Signal-identifier-**set*** of the *Channel-path* in the *Abstract grammar*.

Each <signal identifier> of a <signal list> of a <signal list area> has a corresponding *Signal-identifier* in the *Signal-identifier-**set*** that identifies a *Signal-definition*.

Each <timer identifier> of a <signal list> of a <signal list area> has a corresponding *Signal-identifier* in the *Signal-identifier-**set*** that identifies a *Timer-definition*.

Each <interface identifier> of a <signal list> of a <signal list area> has a corresponding *Signal-identifier* in the *Signal-identifier-**set*** for each *Signal-identifier* of *Signal-identifier-**set*** of the identified *Interface-definition*.

Each *Signal-identifier* in the *Signal-identifier-**set*** appears only once, even if it corresponds to more than one item in the <signal list>.

11 Behaviour

11.1 Start

Abstract grammar

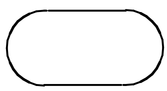
State-start-node :: *Transition*

Concrete grammar

<start area> ::=

<start symbol>
is followed by <transition area>

<start symbol> ::=



Semantics

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

11.2 State

Abstract grammar

State-node :: *State-name*
Save-signalset
*Input-node-**set***
[*Composite-state-type-identifier Connect-node-**set***]

State-name = *Name*

Each *State-node* within a graph (*State-transition-graph* or *Procedure-graph*) shall have *State-name* different from any other *State-node* in the same graph.

A *State-node* without *Composite-state-type-identifier* represents a basic state. A *State-node* with *Composite-state-type-identifier* represents a composite state application. The term "within a composite state" for a state means that the *State-node* for the state is part of the *Composite-state-graph* of an instance of a *Composite-state-type-definition* that is identified by the *Composite-state-type-identifier* of the *State-node*.

The *Connect-node-**set*** shall contain at most one unnamed *Connect-node*.

NOTE 1 – Basic SDL-2010 does not allow a *Connect-node* to have a name, therefore the *Connect-node-**set*** in Basic SDL-2010 contains at most one *Connect-node*.

In the following, *Save-item* and *Input-node* refer, respectively, to a *Save-item* of the *Save-signalset* of a basic *State-node* and an *Input-node* of the *Input-node-**set*** of the same basic *State-node*. If an *Input-node* has no *Gate-identifier*, the *Signal-identifier* of that *Input-node* shall not appear without a *Gate-identifier* in another *Input-node* or a *Save-item*. If an *Input-node* has a *Gate-identifier*, the *Signal-identifier* of that *Input-node* shall not appear with the same *Gate-identifier* in another *Input-node* or a *Save-item*. If a *Signal-identifier* is in the *In-signal-identifier-**set*** of a *Gate-definition* of the state machine that owns the state, the *Signal-identifier* shall occur

a) with the *Gate-identifier* for that gate (in an *Input-node* or a *Save-item*); or

b) without the *Gate-identifier* (in an *Input-node* or a *Save-item*); or

c) if and only if the basic *State-node* is within a composite *State-node*, letting 'composite *Save-item*' refer to a *Save-item* of the *Save-signalset* of the composite *State-node* and letting 'composite *Input-node*' refer an *Input-node* of the *Input-node-**set*** of the same composite *State-node*
   1) with the *Gate-identifier* for that gate in a composite *Input-node* or a composite *Save-item*; or
   2) without the *Gate-identifier* in a composite *Input-node* or a composite *Save-item*; or
   3) if the composite *State-node* is within another composite *State-node* similarly and recursively until a *State-node* of the state machine is reached.

If a *State-node* is in a *Procedure-graph* of a *Procedure-definition*, the *State-node* is within a composite state if and only if the *Procedure-definition* is in the composite state (that is, the *Procedure-definition* is in the *Procedure-definition-**set*** of the *Composite-state-type-definition* for the composite state).

Concrete grammar

<state area> ::=

    <state symbol> ***contains*** <state list>
    is associated with
    {   <input association area>
      | <save association area>
      | <connect association area> }*

<state symbol> ::=

<state list> ::=

    { <basic state name> | <typebased composite state> }

NOTE 2 – Basic SDL-2010 does not include the shorthand feature of multiple names in a state symbol.

<basic state name> ::=

    <state name>

NOTE 3 – The <state name> in a <basic state name> or <composite state name> has to be a <name>, whereas in SDL-2000 it is a name or a number (an <integer name> or a <real name>).

<typebased composite state> ::=
<composite state name> <nextstate parameters>
<colon> <composite state type expression>

<composite state name> ::=
<state name>

NOTE 4 – <typebased composite state> is included for <state area> in Basic SDL-2010 because the *State-machine* of an agent in the abstract syntax has the way it behaves identified by a *Composite-state-type-identifier*.

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

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

In Basic SDL-2010, a <state area> represents a *State-node*.

A <basic state name> is the name of a state that is not defined by a <typebased composite state>. A <composite state name> in a <state list> is the name of a state that is defined by a <typebased composite state>. The <composite state type expression> of a <typebased composite state> in a <state list> identifies the *Composite-state-type-identifier* of the *State-node*.

In Basic SDL-2010 the <state list> contains one <state name>, and the <state name> represents a *State-node*. For each *State-node*, the *Save-signalset* is represented by the <save area> (and any implicit signal saves). For each *State-node*, the *Input-node*-**set** is represented by the <input area> and any implicit input signals.

The <solid association symbol>s originating from a <state symbol> are allowed to have a common originating path.

A <connect association area> is only allowed for a <state area> with <state list> that contains a <typebased composite state>.

A <typebased composite state> of a <state list> shall only contain non-empty <nextstate parameters> if it is in a <state area> that coincides with a <nextstate area>, which is not allowed in Basic SDL-2010.

Semantics

A state represents either a basic state or a composite state application.

The semantics for composite state application is given in clause 11.11.

A basic state represents a particular condition in which the state machine of an agent is able to consume a signal instance. If a signal instance is consumed, the associated transition is interpreted.

For each basic state, the *Save-signalset* and the *Input-node-**set*** are interpreted in the following steps. Each time the steps are repeated, the set of signals considered is updated to the signals on the input port; otherwise, the same set is considered in each step.

a) Signals for inputs that have priority are handled in priority order (in Basic SDL-2010 there is no priority, so no such signals are handled); otherwise

b) in the order of the signals on the input port:

   1) it is evaluated if the current signal is enabled (in Basic SDL-2010 this means checking that the signal is not saved for this state or for the gate of arrival, and the availability time has been reached);

   2) if the current signal is enabled, this signal is consumed for the *Input-node* (see clause 11.3); otherwise

3) if the current state is within a composite state and the current signal is enabled for an *Input-node* (see clause 11.3) of a containing composite state, this signal is consumed for the *Input-node* of the most local such state leaving the composite state; otherwise

4) the next signal on the input port is selected.

c) If no enabled signal was found, any continuous signals are handled (in Basic SDL-2010, continuous signals are not supported so the result will always be to skip to step (d)).

d) If no enabled signal was found, as soon as the available signals on the input port differ from the set of signals already considered, the steps are repeated.

11.3 Input

Abstract grammar

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

The optional *Gate-identifier* shall be a gate of the enclosing state machine: the *Gate-identifier* shall identify a *Gate-definition* of the *Composite-state-type-definition* identified by the *Composite-state-type-identifier* of the *State-machine* for the *Input-node*. The *Gate-definition* shall include the *Signal-identifier* in its *In-signal-identifier-**set***.

The length of the list of optional *Variable-identifier* items shall be the same as the number of *Signal-parameter* items in the *Signal-definition* denoted by the *Signal-identifier*.

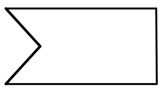
The sorts of the variables shall correspond by position to the sorts of the data items that are carried by the signal.

Concrete grammar

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

<input symbol> ::=
<plain input symbol>

<plain input symbol> ::=



<input list> ::=
<stimulus>

<stimulus> ::=
<signal list item>
[**(** [<variable>] { **,** [<variable>] }* **)**]
[<via path>]

In Basic SDL-2010 the <signal list item> of a <stimulus> shall not represent an interface.

<via path> ::=
via <gate identifier>

NOTE 1 – In Basic SDL-2010 every signal in the valid input signal set has to be mentioned in either an input or a save for a given basic state. In [ITU-T Z.103] the implicit transition feature is introduced as a shorthand notation for consuming any signals not explicitly mentioned.

An <input area> whose <input list> contains one <stimulus> corresponds to one *Input-node*. The <signal identifier> or <timer identifier> contained in the <input symbol> gives the *Signal-identifier* for the *Input-node* that this <input symbol> represents. The <gate identifier> of the optional <via path> of a <stimulus> represents the optional *Gate-identifier* of the *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 clause 11.15.

The <variable> list in <stimulus> represents the *Variable-identifier* list.

NOTE 2 – In [ITU-T Z.103] it is allowed to omit variables, any resulting trailing commas or the complete variable list in the concrete syntax if the values conveyed in the signal are not needed.

A <variable> of a <stimulus> shall not be a global variable of a system (type) or block (type) except if the <stimulus> is within the state machine actions of system (type) or block (type).

Semantics

A signal instance is enabled for an *Input-node* if the signal has the same *Signal-identifier* and if the *Input-node* has a *Gate-identifier* that identifies the gate where the signal arrived, and the current time is greater than or equal to the availability time for the signal instance.

A signal instance is enabled for an *Input-node* if the signal has the same *Signal-identifier* and if the *Input-node* does not have a *Gate-identifier*, and the current time is greater than or equal to the availability time for the signal instance.

If a signal instance is enabled for an *Input-node* that has a *Gate-identifier*, this takes precedence over an *Input-node* that does not have a *Gate-identifier*.

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". Assignment is described in clause 12.3.3. For the assignment the data item of the signal is treated as a *Variable-access* to a variable with the aggregation kind and sort defined by the *Signal-parameter*.

The sender variable of the consuming agent 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 always carry a pid different from any in the system.

11.4 Empty clause

This clause is intentionally left blank.

11.5 Empty clause

This clause is intentionally left blank.

11.6 Empty clause

This clause is intentionally left blank.

11.7 Save

A save specifies a set of signal 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 = *Save-item*-***set***

Save-item = *Signal-identifier* [*Gate-identifier*]

The optional *Gate-identifier* shall be a gate of the enclosing state machine: the *Gate-identifier* shall identify a *Gate-definition* of the *Composite-state-type-definition* identified by the *Composite-state-type-identifier* of the *State-machine* for the *Input-node*. The *Gate-definition* shall include the *Signal-identifier* in its *In-signal-identifier*-***set***.

Concrete grammar

<save area> ::=
<save symbol> ***contains*** { <save list> }

<save symbol> ::=

<save list> ::=
<save item>

<save item> ::=
<signal list item> [<via path>]

A <save list> represents the *Save-signalset*.

Each <signal identifier> of a <signal list item> of a <save list> has a corresponding *Signal-identifier* in a *Save-item* that identifies the *Signal-definition*.

Each <timer identifier> of a <signal list item> of a <save list> has a corresponding *Signal-identifier* in a *Save-item* that identifies the *Timer-definition*.

Each <interface identifier> of a <signal list item> of a <save list> has a corresponding *Save-item* for each *Signal-definition* of *Signal-definition*-***set*** of the identified *Interface-definition*.

The <gate identifier> of the optional <via path> of a <save item> represents an optional *Gate-identifier* of the *Save-item*. In the case of an <interface identifier> as the <save item> with a <via path>, each corresponding *Save-item* has the *Gate-identifier*.

Each *Signal-identifier* in the *Save-signalset* appears only once without a *Gate-identifier*, even if it corresponds to more than one <save item>.

Semantics

A signal in a *Save-signalset* of a state is only enabled for that state if the *Signal-identifier* appears in an *Input-node* of the state; otherwise the signal is saved. If every *Input-node* of a state with the *Signal-identifier* also has a *Gate-identifier*, a *Save-item* without a *Gate-identifier* for that state means that the signal is saved for other gates (if any) for that signal. If there is an *Input-node* of a state with the *Signal-identifier* and no *Gate-identifier*, a *Save-item* for that state with a *Gate-identifier* means that the signal is saved for that gate.

The saved signals are retained in the input port in the order of their availability time and with the arrival gate information as long as the agent remains in the state.

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 that are either consumed or saved in that state.

11.8 Empty clause

This clause is intentionally left blank.

11.9 Empty clause

This clause is intentionally left blank.

11.10 Label (connector name)

Abstract grammar

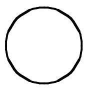
Free-action	::	*Connector-name* *Transition*
Connector-name	=	*Name*

Concrete grammar

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

<connector name> ::=
    <name>
  | <integer name>

<in connector symbol> ::=



The term "body" is used to refer to a state machine graph, possibly after transformation from models. A body in Basic SDL-2010 therefore refers to <procedure body area>, <operation body area> or <composite state body area>.

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

A label <in connector area> is 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.

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 body.

Semantics

A *Free-action* defines the target of a *Join-node*.

11.11 State machine and composite state

A composite state is a state that consists of sequentially interpreted substates (with associated transitions). A substate of a composite state is also a state, and therefore is allowed to be a composite state.

The properties of a composite state (its local substates, transitions, variables and procedures) are defined by its composite state type together with transitions of the state based on the composite state type. These transitions apply to all the substates of the composite state. In Basic SDL-2010 the composite state graph returns control only by an unlabelled return node with interpretation continuing via the unlabelled connect association of the composite state.

11.11.1 Composite state graph

In a composite state graph, the transitions are interpreted sequentially.

Abstract grammar

Composite-state-graph	::	*State-transition-graph*
State-transition-graph	::	[*State-start-node*] *State-node-**set*** *Free-action-**set***

In a specification, all potentially instantiated agents shall have a *State-start-node*. There shall be exactly one unlabelled *State-start-node* in an agent (in the *State-transition-graph* of the *Composite-state-graph* of the *Composite-state-type-definition* identified by the *Composite-state-type-identifier* of the *State-machine* of the *Agent-type-definition* identified by the *Agent-definition*).

A *Composite-state-graph* in Basic SDL-2010 shall have a *State-start-node*.

NOTE – The *State-start-node* is defined as optional in a *State-transition-graph* so that it is possible to omit it in an abstract state type or in a state type that inherits its *State-start-node* from another state type (see [ITU-T Z.102]).

Concrete grammar

```
<composite state structure area> ::=
                {       <composite state text area>*
                        <entity in composite state area>*
                        <composite state body area> }set

<composite state text area> ::=
                <text symbol> contains
                {       <valid input signal set>
                |       <variable definition>
                |       <data definition>}*
```

Each <data definition> of a <composite state text area> represents a member of the *Data-type-definition* set of the *Composite-state-type-definition* if it is a <data type definition> or <interface definition>, and a member of the *Syntype-definition* set of the *Composite-state-type-definition* if it is a <syntype definition>.

A <composite state text area> shall contain a <valid input signal set> only if the corresponding *Composite-state-type-definition* is used for *State-machine* items of agent types. A <composite state text area> containing a <valid input signal set> shall not be used for *Composite-state-type-definition* identified by a composite *State-node*.

```
<entity in composite state area> ::=
                <procedure reference area>
        |       <composite state type reference area>

<composite state body area> ::=
                { <start area>
                        { <state area> | <in connector area> }* } set
```

<composite state body area> represents a *Composite-state-graph*.

Semantics

If a *Composite-state-graph* does not contain a *State-node*, the *Composite-state-graph* is interpreted as an encapsulated part of a transition.

The unlabelled *State-start-node* of the *Composite-state-graph* is the default entry point of the composite state. In Basic SDL-2010 there is no alternative *State-start-node*.

An *Action-return-node* in a composite state 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. In Basic SDL-2010 there is only a *Connect-node* without a *Name*.

The nodes of the state graph of any substate 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.

A composite state is created when the enclosing entity is created, and deleted when the enclosing entity is deleted.

Local variables of the composite state are created and deleted when the composite state is created and deleted respectively.

Each *Composite-state-formal-parameter* is a local variable that is created when the composite state is created. A variable is assigned the result of the expression given by the corresponding actual parameter if present in the *Nextstate-parameters* of a *Nextstate-node* (or *State-machine*) through which the composite state is entered. 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 stimulus as an input or save of the substate.

11.11.2 Empty clause

This clause is intentionally left blank.

11.11.3 Empty clause

This clause is intentionally left blank.

11.11.4 Connect

Abstract grammar

Connect-node :: *Transition*

Concrete grammar

<connect association area> ::=
    <solid association symbol>
    is followed by <exit transition area>

<exit transition area> ::=
    <transition area>

A <connect association area> represents a *Connect-node*.

Semantics

A *Connect-node* represents an exit point on a composite state. Interpretation is resumed at this point if in the *Composite-state-graph* there is interpretation of a *Return-node*.

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*
    | *Set-node*
    | *Reset-node* }

Terminator	::	{ *Nextstate-node* | *Stop-node* | *Return-node* | *Join-node* }

Concrete grammar

<transition area> ::=
[<transition string area> ***is followed by***]
<terminator area>

<terminator area> ::=
<nextstate area>
| <decision area>
| <stop symbol>
| <out connector area>
| <return area>

<transition string area> ::=
<action area>
[***is followed by*** <transition string area>]

<action area> ::=
<task area>
| <output area>
| <create request area>
| <procedure call 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 *Graph-node* list.

A <transition area> in an <operation body area> shall not contain a <state area> or a <nextstate area>.

Semantics

A transition performs a sequence of actions. During a transition, the data of an agent is possibly manipulated and signals possibly output (depending on the content of the transition). The transition ends 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.

It is possible to interpret a transition in one agent of a block at the same time as a transition in another agent of the same block (provided they are not both enclosed by the same 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-2010 system is a complete interleaving of different agents at the level of all actions that it is not possible to transform (by the rules given in the *Model* clauses in SDL-2010 Recommendations) into other actions, and are not excluded because they are in a transition alternating with a transition that is being interpreted (see clause 9.3): that is, waiting for the transition in a containing process to reach a nextstate.

An undefined amount of time passes 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**, is not changed; see clause 12.3.4.1).

11.12.2 Transition terminator

11.12.2.1 Nextstate

Abstract grammar

Nextstate-node	=	*Dash-nextstate* | *Named-nextstate*
Named-nextstate	::	*State-name* [*Nextstate-parameters*]
Nextstate-parameters	::	*Actual-parameters*
Dash-nextstate	::	[**HISTORY**]

Nextstate-parameters shall only be present if *State-name* denotes a composite state.

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

Every *Transition* that terminates in a *Dash-nextstate* shall originate from a *State-node* within the same *State-transition-graph* or *Procedure-graph*, either directly or via *Decision-node* items or *Join-node* items and *Free-action* items.

NOTE 1 – For example, it is not allowed to have a *Dash-nextstate* reachable from a *State-start-node* without interpretation of a *State-node*.

If a *Named-nextstate* includes *Nextstate-parameters*, the *State-name* shall refer to a composite state (a *State-node* with a *Composite-state-type-identifier*). In that case, the *Actual-parameters* shall have the same number of elements as the *Composite-state-formal-parameter* list of the identified *Composite-state-type-definition*. Each *Expression of* the *Actual-parameters* shall have a sort that is compatible with the sort of the corresponding by position *Composite-state-formal-parameter* in the *Composite-state-type-definition*.

Concrete grammar

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

<nextstate body> ::=
                    <nextstate body name>
               |    <dash nextstate>
               |    <history dash nextstate>

<nextstate body name> ::=
                    <basic state name>
               |    <composite state name> <nextstate parameters>

<nextstate parameters> ::=
                    [<actual parameters>]

<dash nextstate> ::=
                    <hyphen>

<history dash nextstate> ::=
                    <history dash sign>
```

A <history dash nextstate> represents a *Nextstate-node* that is a *Dash-nextstate* with **HISTORY**. A <dash nextstate> represents a *Nextstate-node* that is a *Dash-nextstate* without **HISTORY**.

Semantics

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

For a *Named-nextstate* the state is the *State-node* within the same *State-transition-graph* or *Procedure-graph* that has the *State-name* of the *Named-nextstate*. If the *State-name* refers to a composite state (a *State-node* with a *Composite-state-type-identifier*), each *Expression* of the

Nextstate-parameters is interpreted and assigned to the corresponding by position *Composite-state-formal-parameter*. If the sort of a *Composite-state-formal-parameter* is a syntype, a range check is performed as further described under *Semantics* in clause 12.2.1. For each UNDEFINED element of the *Nextstate-parameters* the corresponding by position *Composite-state-formal-parameter* has no data associated: that is, it is "undefined". If *Nextstate-parameters* are omitted, each *Composite-state-formal-parameter* has no data associated: that is, it is "undefined".

For a *Dash-nextstate*, the activating state is the *State-node* within the same *State-transition-graph* or *Procedure-graph* that activated the current *Transition*, or – if that *Transition* was activated by another *Transition* (for example, a *Transition* ending in *Decision-node*) – the activating state of that *Transition*.

An empty *Dash-nextstate* means that the activating state is entered again. An empty *Dash-nextstate* for a composite state implies that the next state is the composite state, and is entered in the same way as a *Named-nextstate* for that composite state without *Nextstate-parameters*.

NOTE 2 – If there is only one state that leads to the *Dash-nextstate*, the *Dash-nextstate* has the same meaning as a *Nextstate-node* that has the *State-name* of this state.

When a *Dash-nextstate* with **HISTORY** is interpreted, the next state is the activating state, or a state within the activating state, if the activating state is a composite state. If the activating state is not a composite state, the meaning is the same as an empty *Dash-nextstate*. If a composite state is re-entered, the next state is the last state in the composite state (if any) before the exit from the composite state. If this state was itself a composite state this inner composite state is re-entered in the same way. If there was no last state in the composite state, the composite state is re-entered in the same way as an empty *Dash-nextstate*.

NOTE 3 – In Comprehensive SDL-2010 if interpretation re-enters a composite state, its entry procedure if it exists is invoked. Entry procedures are not a feature of Basic SDL-2010.

When determining the activating state, any procedure call is ignored, even if the procedure contains internal states.

NOTE 4 – In Comprehensive SDL-2010 implicit states exist for items such a remote procedure calls. These are also treated as encapsulated in procedure calls and therefore are not considered for the activating state.

11.12.2.2 Join

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

Abstract grammar

Join-node :: *Connector-name*

Concrete grammar

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

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

For each <out connector area> in a body area (such as <composite state body area>, <operation body area> or <procedure body area>), there shall be exactly one <in connector area> in that body area with the same <connector name>.

NOTE – In [ITU-T Z.103] it is possible to join two transitions with a merge area which is transformed into <out connector area>s and an <in connector area>.

Semantics

When a *Join-node* is interpreted, interpretation continues with the *Free-action* named with *Connector-name*.

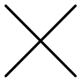
11.12.2.3 Stop

Abstract grammar

Stop-node :: { }

Concrete grammar

<stop symbol> ::=



A <stop symbol> represents a *Stop-node*.

Semantics

If the number of instances in the agent instance set is already at the *Lower-bound* for that instance set, the predefined exception `OutOfRange` is raised.

NOTE – To avoid `OutOfRange` being raised, it is possible to use an active agents expression (see clause 12.3.4.4) to determine if the number of instances is already at the *Lower-bound*.

If `OutOfRange` is not raised, the stop causes the agent interpreting it to perform a stop.

This means that the retained stimuli in the input port (other than the implicit set and get remote procedure calls, if any, introduced for each global variable as described in [ITU-T Z.102], [ITU-T Z.103] and [ITU-T Z.104]) 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.

The interpretation of a *Stop-node* in a *Procedure-graph* or *State-transition-graph* causes the agent interpreting that g*raph* to stop. Interpretation of the procedure (or operation, or composite state if appropriate) terminates, and the stop propagates outwards to the caller and is treated as if a *Stop-node* were interpreted at the place of the procedure call (or operation application, or entry to the composite state if appropriate). Termination propagates outwards until the containing agent is reached.

11.12.2.4 Return

Abstract grammar

Return-node = *Action-return-node*
| *Value-return-node*

Action-return-node :: { }

Value-return-node :: *Expression*

An *Action-return-node* shall only be contained in the *Procedure-graph* of a *Procedure-definition* without *Result* or a *Composite-state-graph*. A *Value-return-node* shall only be contained in the *Procedure-graph* of a *Procedure-definition* containing *Result*.

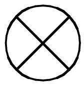
The *Expression* of a *Value-return-node* shall be sort compatible with the sort of the *Result* of the enclosing *Procedure-definition*.

Concrete grammar

<return area> ::=
<return symbol>
is associated with <return body>

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

<return symbol> ::=



A <return area> with an empty <return body> represents an *Action-return-node*. A <return area> with an <expression> for a <return body> represents a *Value-return-node*.

An <expression> that is a <return body> in <return area> is allowed if and only if the enclosing scope is an operator (or method; see [ITU-T Z.104]), or a procedure that has a <procedure result>. The <expression> that is a <return body> in <return area> shall not be omitted if the enclosing scope is an operator (or method; see [ITU-T Z.104]) with an <operation result>, or a value returning procedure with a <procedure result> without a <variable name> (a <procedure result> never has a <variable name> in Basic SDL-2010).

Semantics

A *Return-node* in a procedure is interpreted in the following ways.

a) If a *Value-return-node* is interpreted, the result of *Expression* is interpreted in the same way as an *Expression* assigned to a variable with the sort of the result (see clause 12.3.3), but without the result being associated with a variable or a range check taking place and the result is returned for use in the calling context.

b) All variables created by the interpretation of the *Call-node* or *Value-returning-call-node* cease to exist.

c) The interpretation of the *Procedure-graph* is completed and the procedure instance ceases to exist.

d) Interpretation of the calling context continues.

An *Action-return-node* in the composite state that is the state machine of an agent is interpreted as a *Stop-node*.

An *Action-return-node* (a *Value-return-node* is not allowed) in a composite state that is not the state machine of an agent results in activation of a *Connect-node*, and interpretation continues at the *Connect-node* without a name.

11.13 Action

11.13.1 Task

Abstract grammar

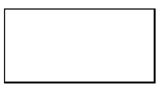
Task-node	=	*Assignment*
	|	*Informal-text*

Concrete grammar

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

<task body> ::=
<non terminating statements> <end>*
| <informal text>

<task symbol> ::=



In Basic SDL-2010 <non terminating statements> of <task body> of <task area> is a single <statement> that is an <assignment statement>, <set statement> or <reset statement>. A <task area> containing a single <assignment> represents an *Assignment* in the *Task-node*. A <task area>

containing a single <set statement> corresponds to a *Set-node* as an element of the *Graph-node* list for the *Transition* of <transition area> containing the <task symbol>. A <task area> containing a single <reset statement> corresponds to a *Reset-node* as an element of the *Graph-node* list for the *Transition* of <transition area> containing the <task symbol>.

Semantics

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

11.13.2 Create

Abstract grammar

Create-request-node :: { *Agent-identifier* | **THIS** }
Actual-parameters

The length of the *Actual-parameters* list shall be the same as the number of *Agent-formal-parameter* items in the *Agent-definition* of the *Agent-identifier*.

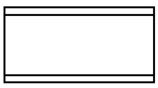
Each *Expression* of the *Actual-parameters* corresponding by position to an *Agent-formal-parameter* shall have a sort that is compatible with the sort of the *Agent-formal-parameter* in the *Agent-definition* denoted by *Agent-identifier*, or the local *Agent-type-definition* if **THIS** is given.

THIS shall only be specified in an *Agent-type-definition* and in scopes enclosed by an *Agent-type-definition*.

Concrete grammar

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

<create request symbol> ::=



<create body> ::=
{ <agent identifier> | **this** } [<actual parameters>]

NOTE – The alternative of an <agent type identifier> is not allowed in Basic SDL-2010.

this represents **THIS**.

A <create body> of a <create request area> represents a *Graph-node* that is a *Create-request-node*.

Semantics

The create action causes the creation of an instance of the set identified by *Agent-identifier* either inside the agent that performs the create, or in an agent that contains the agent that performs the create. The **parent** of the created agent (see clause 9) has the same pid as returned by **self** of the creating agent. **self** of the created agent (see clause 9) and **offspring** of the creating agent (see clause 9) have both the same unique, new pid.

When an agent instance is created, it is given an empty input port, and variables are created. The creating agent **offspring** is set and the actual parameter expressions are interpreted and assigned to the corresponding formal parameters, and if the sort of a formal parameter is a syntype, a range check is made as further described under *Semantics* in clause 12.2.1. If the created agent has contained agent sets, then the initial instances of these sets (if any) are created with **parent** as `Null`. 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) has the result `Null` and interpretation continues.

If an *Expression* in *Actual-parameters* is **UNDEFINED**, the corresponding formal parameter has no data item associated; that is, it is "undefined". If `OutOfRange` is raised assigning a parameter, the creation continues, but the remaining parameter has no data associated with it as if the corresponding *Actual-parameters* element was **UNDEFINED**.

THIS identifies the *Agent-identifier* of the set of instances of the agent in which the create is being interpreted.

11.13.3 Procedure call

Abstract grammar

Call-node :: *Procedure-identifier* *Actual-parameters*

Value-returning-call-node :: *Procedure-identifier* *Actual-parameters*

The *Procedure-identifier* shall identify *Procedure-definition* with a *Procedure-graph* that has a *Procedure-start-node*.

The length of the *Actual-parameters* list shall be the same as the number of the *Procedure-formal-parameter* items in the *Procedure-definition* denoted by the *Procedure-identifier*.

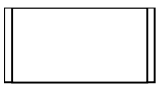
Each *Expression* in the *Actual-parameter*s list corresponding by position to an *In-parameter* shall be sort compatible with the sort of the *Procedure-formal-parameter*.

Each element in the *Actual-parameter*s list corresponding by position to an *Inout-parameter* or *Out-parameter* shall be an *Expression* for a *Variable-identifier* with the same *Sort-reference-identifier* as the *Procedure-formal-parameter*.

Concrete grammar

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

<procedure call symbol> ::=



<procedure call body> ::=
<procedure type expression> [<actual parameters>]

NOTE 1 – In Basic SDL-2010 <procedure type expression> is limited to <base type>, which is a <procedure identifier>. In [ITU-T Z.102] actual context parameters are allowed in <type expression> and in this case the *Procedure-identifier* identifies an implicitly created procedure definition.

A <procedure call area> represents a *Call-node*. A <value returning procedure call> (see clause 12.3.5) represents a *Value-returning-call-node*.

Semantics

The interpretation of a procedure *Call-node* or *Value-returning-call-node* interprets the actual parameter expressions in the order given. If no exceptions are raised by the parameter interpretation, interpretation is then transferred to the procedure definition referenced by the *Procedure-identifier*, and that procedure graph is interpreted (the explanation is contained in clause 9.4).

If an *Expression* in *Actual-parameter*s is omitted, the corresponding formal parameter has no data item associated; that is, it is "undefined".

If an argument sort of the *Call-node* or *Value-returning-call-node* for an *In-parameter* of the procedure is a syntype, the range check defined in clause 12.1.8.2 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` is raised instead of interpreting further actual parameters or the procedure definition.

If `OutOfRange` is not raised, the interpretation of the transition containing a *Call-node* continues when the interpretation of the called procedure is finished.

If `OutOfRange` is not raised, 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*.

The (static) sort of the *Value-returning-call-node* is the sort identified by the *Result* of the *Procedure-definition* identified by the *Procedure-identifier*. The aggregation kind of a *Value-returning-call-node* is the *Result-aggregation* of the *Procedure-definition* identified by the *Procedure-identifier*.

If the result sort of a value returning procedure call is a syntype, the range check defined in clause 12.1.8.2 is applied to the result of the procedure call. If the range check is the predefined `Boolean` value `false` at the time of interpretation, then the predefined exception `OutOfRange` is raised.

Model

If the procedure identified by the <procedure type expression> of the <procedure call body> is not defined within the agent type enclosing the call, within the enclosing agent type there is an implicitly defined local procedure with the same name as identified by the <procedure type expression> and the call uses this local procedure. In the local procedure, identifiers of items (such as variables) external to the procedure definition are bound in the context of the original procedure definition rather than the context of the procedure call if that is different.

NOTE 2 – An implicitly defined local procedure is an inherited subtype of the procedure identified by the <procedure type expression> of the <procedure call body> (see clause 8.4 Specialization of [ITU-T Z.102], and clause 9.4 Procedure of [ITU-T Z.102]).

11.13.4 Output

Abstract grammar

Output-node	::	*Signal-identifier* *Actual-parameters* *Activation-delay* *Signal-priority* [*Signal-destination*] *Direct-via*
Activation-delay	=	*Expression*
Signal-priority	=	*Expression*
Signal-destination	::	{ *Expression* | *Agent-identifier* | **THIS** } [*Destination-number*]
Destination-number	=	*Expression*
Direct-via	=	*Gate-identifier**-set***

The *Signal-identifier* shall identify a *Signal-definition*.

The length of the *Actual-parameters* list shall be the same as the number of *Signal-parameter* items in the *Signal-definition* denoted by the *Signal-identifier*.

Each *Expression* of the *Actual-parameters* list shall be sort compatible with the *Sort-reference-identifier* of the corresponding (by position) *Signal-parameter* in the *Signal-definition.*

The *Expression* of the *Activation-delay* shall be a Duration expression.

The *Expression* of the *Signal-priority* shall be a Natural expression.

For each *Gate-identifier* in *Direct-via,* there shall 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 shall include the *Signal-identifier*.

The sort of *Expression* of a *Signal-destination* shall either be the sort Pid (see clause 12.1.5), or the sort *Interface-definition* with the *Signal-identifier* in its *Signal-identifier-**set***. The *Destination-number* is always omitted for a *Signal-destination* that is an *Expression*.

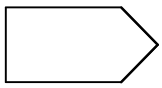
The *Agent-identifier* of a *Signal-destination* shall identify an agent instance set reachable from the originating agent. The *Expression* of the *Destination-number* shall be a Natural expression.

Concrete grammar

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

<output symbol> ::=
<plain output symbol>

<plain output symbol> ::=



<output body> ::=
<output body item>
<communication constraints>

<output body item> ::=
<signal identifier> [<actual parameters>] [<activation delay>] [<signal priority>]

<communication constraints> ::=
{ **to** <destination> | <via path> }*

<destination> ::=
<pid expression0>
| { [**system** | **block** | **process**] <agent identifier> | **this** } [<destination number>]

<destination number> ::=
<left square bracket> <Natural expression0> <right square bracket>

<activation delay> ::=
active <Duration expression>

<signal priority> ::=
priority <Natural expression>

The <pid expression0> or the <agent identifier> in <destination> represents the *Signal-destination.* There is a syntactic ambiguity between <pid expression0> and <agent identifier> in <destination>. If it is possible to interpret the <destination> as a <pid expression0> without violating any static conditions, it is interpreted as <pid expression0>, otherwise as <agent identifier>.

Signals mentioned in <output body>s of the state machine of an agent type shall 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.

In Basic SDL-2010, a <communication constraints> shall contain at most one **to** <destination> clause.

Each <via path> of <communication constraints> represents a *Gate-identifier* in the *Direct-via*.

The optional keyword (**system**, **block** or **process**) before an <agent identifier> in <destination> shall match the *Agent-kind*.

this represents **THIS**.

If <activation delay> is omitted, the *Activation-delay* is zero: that is, there is no delay in activating the signal at the destination.

If <signal priority> is omitted, the *Signal-priority* is zero: that is, the signal has the highest signal priority.

Semantics

Stating an *Agent-identifier* with no *Destination-number* in *Signal-destination* indicates *Signal-destination* is the pid value of any existing instance of the set of agent instances indicated by *Agent-identifier*. If no instances exist, the signal is discarded.

Stating **THIS** with no *Destination-number* in a *Signal-destination* is the pid value of one the set of instances of the agent in which the output is being interpreted.

Stating an *Agent-identifier* or **THIS** with a *Destination-number* in *Signal-destination* indicates *Signal-destination* is the pid of the indicated instance of the set of agent instances. The agent instances are numbered consecutively from 1 when the *Signal-destination* or **THIS** is interpreted in the order in which the instances were created: this allows for changes in numbering due to instances terminating. If *Destination-number* is zero or greater than the number of instances in the set of agent instances, the signal is discarded.

If no *Gate-identifier* is specified in *Direct-via* and no *Signal-destination* is specified, any agent for which there exists a communication path is able to receive the signal.

If there is a process instance that contains both the sender and the receiver, 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. Each conveyed data item is equal to the corresponding actual parameter of the output.

NOTE 1 – For Basic SDL-2010 replicates of the results of the actual parameters of the output are the same as results of the actual parameters, so there is no difference in the information passed in the two cases. The distinction is only relevant for parameters that contain elements to identify created data items, as existed with **object** data types in SDL-2000.

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". Otherwise the *Expression* is assigned to the parameter of the signal as if this is a variable (see clause 12.3.3) with the aggregation kind and sort as defined by the *Signal-parameter*.

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 clause 12.1.8.2 is applied to the expression.

If *Signal-destination* is an *Expression* and the static sort of the pid expression is Pid, then the output compatibility check (see clause 12.1.5) is performed for the destination given by the pid expression and the signal denoted by the *Signal-identifier*.

The signal instance is then delivered to a communication path able to convey it. It is possible to restrict the set of communication paths able to convey the signal 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.

When the output is interpreted, the *Activation-delay* is added to the current value of **now** to determine the availability Time: that is, the time after which the signal is made available in the input port of the destination. If the *Activation-delay* is positive the signal instance contains the availability Time; otherwise the signal instance is sent without this information.

If the *Signal-priority* is non-zero, the value is conveyed with the signal instance to the destination; therefore a zero signal priority is implied for any signal instance that does not contain a signal priority value.

If no *Signal-destination* is specified, the receiver is selected in two steps. First, the signal is sent to an agent instance set, which is reachable 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 is selectable, the signal instance is discarded.

When a signal instance is delivered to an instance of an agent instance set and there is an internal communication path that conveys the signal to the state machine of the agent instance, the signal instance is delivered to that state machine. Otherwise, a communication path within the agent instance able to convey the signal instance is arbitrarily chosen and the signal instance is delivered to an instance set of a contained agent.

NOTE 3 – Specifying the same *Gate-identifier* in the *Direct-via* of two different *Output-node* occurrences does not necessarily mean that the signals are queued in the input port in the same order as the *Output-node* occurrences are interpreted. However, order is preserved if the paths conveying the two signals are identical (that is, the signals take the same route), or they are only conveyed on paths with no delay. If the first or both signals have a positive *Activation-delay*, the order the signals are queued in the input port depends on the time the signals are received and the calculated availability time of each signal instance.

11.13.5 Decision

Abstract grammar

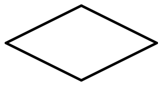
Decision-node	=	*Decision-body*
Decision-body	::	*Decision-question* *Decision-answer-**set*** [*Else-answer*]
Decision-question	=	*Expression*
	|	*Informal-text*
Decision-answer	::	{ *Range-condition* | *Informal-text* } *Transition*
Else-answer	::	*Transition*

Each *Constant-expression* of the *Range-condition* shall be sort compatible with the sort of the *Decision-question*. If the *Decision-question* is an *Expression*, the *Range-condition* of each *Decision-answer* shall be sort compatible with the sort of the *Decision-question*.

Concrete grammar

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

<decision symbol> ::=



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

<answer part> ::=
<transition area> ***is associated with*** <graphical answer>

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

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

<else part> ::=
<transition area> ***is associated with*** **else**

The <graphical answer> and **else** are placed alongside the <flow line symbol>, or over the <flow line symbol> leading to the <transition area> of the <answer part> or <else part>, respectively. In the diagram ***is followed by*** is shown by <flow line symbol> between the <decision symbol> and <answer part> or <else part>.

The <flow line symbol>s originating from a <decision symbol> are allowed to have a common originating path (that is, a part of the <flow line symbol>s overlap), but each <flow line symbol> shall also have some distinct part leading to the <transition area>. The <graphical answer> or **else** are placed sufficiently close to this distinct part of the line so that the association with the <answer part> or <else part> is unambiguous, or is placed over the line. Each <flow line symbol> originates from the left or bottom or right vertex of the <decision symbol>. It is not required that each <flow line symbol> originates from the same vertex.

A <decision area> represents a *Decision-node*. A <question> represents a *Decision-question* of a *Decision-body* and the following <answer part> set and optional <else part> represent the *Decision-answer-**set*** and optional *Else-answer*, respectively. The <range condition> or <informal text> of the <answer> represent the *Range-condition* or *Informal-text* of the *Decision-answer*. The <transition area> of the <answer part> represents the *Transition* of the *Decision-answer*.

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 treated as <informal text>. If the <question> is a <character string> or any <answer> is a <character string> and this 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 that are <character string>s.

Semantics

A *Decision-body* transfers the interpretation to the *Transition* of the outgoing *Decision-answer*, whose *Range-condition* contains the result given by the interpretation of the question, or if there is no such *Decision-answer* to the *Else-answer* (if there is one). The determination of whether the *Decision-question* is contained in each *Decision-answer* is carried out once for each *Decision-answer* in an arbitrary order until a *Range-condition* containing the *Decision-question* is identified, or until this determination requires interpretation of an operation application that raises an exception, or an *Informal-text* is chosen.

The *Else-answer* indicates the *Transition* to be interpreted 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 *Decision-question* does not match any *Decision-answer*, the predefined exception NoMatchingAnswer is raised.

11.14 Statement lists

In this Recommendation statement list is limited to one statement, but in [ITU-T Z.102] the syntax is extended to include additional kinds of statements and extended in [ITU-T Z.103] to include further shorthand notations.

Concrete grammar

```
<non terminating statements> ::=
                        <non terminating statement>

<non terminating statement> ::=
                        <statement>

<statement> ::=
                        <assignment statement>
                  |     <set statement>
                  |     <reset statement>

<assignment statement> ::=
                        <assignment>
```

An <assignment statement> represents an *Assignment* in a *Task-node*.

```
<set statement> ::=
                        set <set body>
```

A <set statement> represents a *Set-node* (see clause 11.15).

```
<reset statement> ::=
                        reset <reset body>
```

A <reset statement> represents a *Reset-node* (see clause 11.15).

11.15 Timer

Abstract grammar

Timer-definition	::	*Timer-name* *Sort-reference-identifier** [*Timer-default-initialization*]
Timer-default-initialization	=	*Constant-expression*
Timer-name	=	*Name*
Set-node	::	*Time-expression* *Timer-identifier* *Expression**
Reset-node	::	*Timer-identifier* *Expression**
Timer-identifier	=	*Identifier*
Time-expression	=	*Expression*

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

The number of items of the *Expression* list in the *Set-node* and *Reset-node* shall be the same as the number of items in the *Sort-reference-identifier* list directly following the *Timer-name* in the *Timer-definition* identified by the *Timer-identifier*.

Concrete grammar

<timer definition> ::=

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

<timer definition item> ::=

<<u>timer</u> name> [<sort list>] [<timer default initialization>]

NOTE 1 – The <<u>timer</u> name> has to be a <name>, whereas in SDL-2000 it is a name or a number (an <integer name> or a <real name>).

Each <aggregation kind> in the <sort list> of a <timer definition item> shall be empty.

NOTE 2 – To simplify the syntax <sort list> is used rather than introduce a timer sort list, but aggregation kind is not relevant for timers.

Each <timer definition item> represents a Timer-definition.

Each <sort> in the <sort list> of a <timer definition item> adds the Sort-reference-identifier to the end of the Sort-reference-identifier list of the Timer-definition.

<timer default initialization> ::=

<is assigned sign> <<u>Duration</u> constant expression>

A <timer default initialization> represents the optional *Timer-default-initialization*.

<reset body> ::=

(<reset clause> **)**

<reset clause> ::=

<<u>timer</u> identifier> [**(** <expression list> **)**]

<set body> ::=

<set clause>

<set clause> ::=

([<<u>Time</u> expression> **,**] <<u>timer</u> identifier> [**(** <expression list> **)**] **)**

A <set clause> with a <<u>Time</u> expression> represents a *Set-node* where the <<u>Time</u> expression> represents the *Time-expression*.

If a <set clause> has no <<u>Time</u> expression>, the *Time-expression* of the *Set-node* is the time value for **now** + the value of the *Timer-default-initialization* of the identified *Timer-definition*.

NOTE 3 – That is: a <set clause> with no <<u>Time</u> expression> is equivalent to a <set clause> where <<u>Time</u> expression> is:

now + <<u>Duration</u> constant expression>

where <<u>Duration</u> constant expression> is derived from the <timer default initialization> in the timer definition.

It is allowed to omit <<u>Time</u> expression> in a <set clause>, only if the identified *Timer-definition* has a *Timer-default-initialization*.

Semantics

A *Timer-definition* defines both the type of a timer and a set of timer instances. A *Signal-identifier* for a signal instance put into the input port of the agent owning the timer identifies the *Timer-definition*. When a *Timer-definition* has an empty *Sort-reference-identifier* list, there is only one timer instance; otherwise there are as many timer instances as there are possible different actual values for the *Expression* list in *Set-node* interpretations for the *Timer-identifier*.

NOTE 4 – The number of timer instances for a timer with parameters is in principle unbounded if there are an unbounded possible number of values for the *Expression* list. However, in practice in a finite run of the system it is necessary to include in the implementation of the set of timer instances only those that have been set, and treat any other timer instance as inactive.

A timer instance is 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 clause 12.2.4) 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 instance with a *Signal-identifier* that identifies the *Timer-definition* 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 clause 12.1.8.2 applied to the corresponding expression in a set or reset shall 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, the timer is inactive.

The *Expression* items in a *Set-node* or *Reset-node* are evaluated in the order given, left to right.

12 Data

The concept of basic data in SDL-2010 is defined in this clause. Data are more fully defined in [ITU-T Z.104] and is further extended for object-oriented data in [ITU-T Z.107].

12.1 Data definitions

Data definitions are used to define data types. The basic mechanisms to define data are data type definitions (see clause 12.1.1) and interfaces (see clause 12.1.2). The definition of the sort of a data type (as well as operations implied for the sort) are given by data type constructors (see clause 12.1.6). Additional operations are defined as described in clause 12.1.3. The way to define behaviour of the operations of a data type is described in clause 12.1.7.

Since predefined data are defined in a predefined and implicitly used package `Predefined` (see clause 7.2), the predefined sorts (for example, `Boolean` and `Natural`) and their operations are available to be freely used throughout the system. The semantics of Equality (clause 12.2.4), Conditional expressions (clause 12.2.5), and Syntypes (clause 12.1.8.1) rely on the definition of the `Boolean` data type.

Abstract grammar

Data-type-definition	=	*Value-data-type-definition*
	|	*Interface-definition*
Value-data-type-definition	::	*Sort* [*Data-type-identifier*] *Literal-signature-**set*** *Static-operation-signature-**set*** *Procedure-definition-**set*** *Data-type-definition-**set*** *Syntype-definition-**set***

[*Default-initialization*]

Interface-definition	::	*Sort* *Null-literal-signature* *Data-type-identifier-**set*** *Signal-definition-**set*** *Signal-identifier-**set***
Null-literal-signature	=	*Literal-signature*
Data-type-identifier	=	*Identifier*
Sort-reference-identifier	= |	*Sort-identifier* *Syntype-identifier*
Sort-identifier	=	*Identifier*
Sort	=	*Name*

A *Data-type-definition* introduces a sort that is visible in the enclosing scope unit in the abstract syntax. It additionally and optionally introduces a set of literals and/or operations.

The *Data-type-identifier* of a *Value-data-type-definition* is omitted for data types (such as those defined by Basic SDL-2010) that do not use inheritance.

Each *Procedure-definition* of the *Procedure-definition-**set*** of a *Value-data-type-definition* is a *Procedure-definition* associated with an *Operation-signature* according to clause 12.1.7.

Concrete grammar

```
<data definition> ::=
                <entity in data type>
        |       <interface definition>
```

The *Default-initialization* of a *Value-data-type-definition* is included only for an <entity in data type> that is a <data type definition> including a <default initialization> (see clauses 12.1.1 and 12.3.3.2).

A <data definition> represents a member of the *Data-type-definition* set of the enclosing entity if it is an <interface definition>, or an <entity in data type> that is a <data type definition>, or a member of the *Syntype-definition* set of the enclosing entity if it is an <entity in data type> that is a <syntype definition>.

```
<sort> ::=
                <basic sort>
        |       <pid sort>

<basic sort> ::=
                <datatype type expression>
        |       <syntype>

<pid sort> ::=
                <sort identifier>
```

A <<u>sort</u> identifier> identifies a sort (a set of elements or data items) introduced by a data type definition.

Each <data type definition> introduces a sort with the same name as the <<u>data type</u> name> (see clause 12.1.1). Each <interface definition> introduces a sort with the same name as the <<u>interface</u> name> (see clause 12.1.2).

NOTE – To avoid cumbersome text, the convention is used that the phrase "the sort S" (or "the S sort") is often used in SDL-2010 Recommendations 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.

The <<u>sort</u> identifier> in a <pid sort> shall identify a pid sort: that is, a *Sort* introduced by an *Interface-definition*.

Semantics

A data definition is used either for the definition of a data type or interface. A *Value-data-type-definition* introduces a *Sort* that is a set of values as further defined in clause 12.1.1, and an *Interface-definition* introduces a pid sort as further defined in clause 12.1.8.1.

A sort is a set of elements: values or pids (that is, identities of agents). Two different sorts have no elements in common.

The *Data-type-identifier* of a *Value-data-type-definition* identifies the base data type (super type) of the data type. In Basic SDL-2010 this is always omitted. There are generic operations that apply to all data types, such as `equal` that compares two values for equality, and other generic operations that apply to all literal, or all structure or all choice sorts.

NOTE – Specialization (as defined in [ITU-T Z.102] and not included in Basic SDL-2010) allows the *Data-type-identifier* of the base type to be given.

A *Data-type-name* used as a *Data-type-qualifier* to identify a *Value-data-type-definition* as a scope unit has the same *Name* as the *Sort* of the *Value-data-type-definition*.

An *Interface-name* used as an *Interface-qualifier* to identify an *Interface-definition* as a scope unit has the same *Name* as the *Sort* of the *Interface-definition*.

12.1.1 Data type definition

A data type definition has a body that usually contains a data type constructor.

The data type constructor defines how to construct sets of values (structured values, literal values and choice values). If the data type definition is a **value** type, these values are the elements of the sort.

Concrete grammar

<entity in data type> ::=
<data type definition>
| <syntype definition>

<data type definition> ::=
{<package use clause>}*
<type preamble> <data type heading>
{ <end>
| [<comment body>] <left curly bracket> <data type definition body>
<right curly bracket> }

A <comment body> is a form of annotation and has no formal semantic meaning.

<data type definition body> ::=
{<entity in data type>}* [<data type constructor>] <operations>
[<default initialization> <end>]

A <data type constructor> (see clause 12.1.6) describes the elements of the sort and operations included by the way the sort is composed. The <operations> defines a set of operations for elements of a sort (see clause 12.1.3 and clause 12.1.7).

<data type heading> ::=
value type <data type name>

NOTE – The <data type name> has to be a <name>, whereas in SDL-2000 it is a name or a number (an <integer name> or a <real name>).

A <data type name> represents the *Sort* of the *Data-type-definition*, and this <name> is the *Data-type-qualifier* as the <name> of a <path item> for a <qualifier> to identify the *Data-type-definition* as a scope unit.

<operations> ::=
<operation signatures>
<operation definitions>

A <value data type definition> contains the keyword **value** in <data type heading> and represents a *Value-data-type-definition*.

For each <operation signature> of <operation signatures>, there shall be one and only one corresponding <operation reference> in the <operation definitions> of the <operations>.

12.1.2 Interface definition

Interfaces are defined in packages, agents or agent types.

An interface definition defines the set of signals for the interface. These signals include the signals defined within the interface and signals defined outside the interface included through a list of interfaces used in the interface. An interface definition introduces a pid sort, which has elements that are identities of agents that handle the signals for the interface. 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 is used in a signal list to denote that the signals of the interface definition are included in the signal list.

Abstract grammar

Interface-definition is defined in clause 12.1.

Each *Data-type-identifier* of the *Data-type-identifier-**set*** of an *Interface-definition* shall identify an interface, or there shall be only one item in the *Data-type-identifier-**set*** of the *Interface-definition* and this shall identify the data type `Pid`.

NOTE 1 – For Basic SDL-2010 the *Data-type-identifier-**set*** of an *Interface-definition* always has just one `Pid` item.

Concrete grammar

Each agent type (and agent and state machine) implicitly defines an *Interface-definition* as detailed below. This *Interface-definition* is in the same context as the definition of the agent type (or agent or state machine), so that (for example) the implicit *Interface-definition* for an item of the *Agent-type-definition-**set*** of an *Agent-type-definition* is an item of the *Data-type-definition-**set*** of the *Agent-type-definition*.

Interfaces are implicitly defined by each agent type definition and each agent definition (except the outermost agent) and by the state machine of each agent type definition. The implicitly defined interface for an agent or an agent type has the same name and is defined in the same scope unit as the agent or agent type that defined it. The implicitly defined interface for a state machine has the same name as the containing agent type but is defined in the same scope unit as the state machine that defined it: that is, inside the agent type.

<interface definition> ::=
{<package use clause>}*
<interface heading>
{ [<comment body>]
<left curly bracket>
<entity in interface>*
<right curly bracket>}

A <comment body> is a form of annotation and has no formal semantic meaning.

<interface heading> ::=
interface <interface name>

<entity in interface> ::=
<signal definition list>
| <interface use list>

<interface use list> ::=
use <signal list> <end>

Each <signal list item> of the <signal list> in an <interface use list> of an <interface definition> shall be a <signal identifier> or an <interface identifier>. An <interface identifier> that is part of the <signal list> shall also respect the restriction.

The <interface definition> shall not contain the <interface identifier> defined by the <interface definition> either directly or indirectly (via another <interface identifier>).

An <interface name> represents the *Sort* of the *Interface-definition*, and this <name> is the *Interface-qualifier* as the <name> of a <path item> for a <qualifier> to identify the *Interface-definition* as a scope unit.

NOTE 2 – The <interface name> in the heading has to be a <name>, whereas in SDL-2000 it is a name or a number (an <integer name> or a <real name>).

NOTE 3 – 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.

NOTE 4 – An <interface use list> does not define entities.

In Basic SDL-2010 the *Data-type-identifier* set of the *Interface-definition* for a pid sort contains only the data type `Pid`, which is further defined in clause 12.1.5 and fully defined [ITU-T Z.104].

A <signal list> of an <interface use list> denotes items in the *Signal-identifier-**set*** of the *Interface-definition* as follows.

a) Each <signal identifier> of a <signal list> of an <interface use list> represents a corresponding *Signal-identifier* in the *Signal-identifier-**set*** that identifies a *Signal-definition*.

b) Each <timer identifier> of a <signal list> of an <interface use list> represents a corresponding *Signal-identifier* in the *Signal-identifier-**set*** that identifies a *Timer-definition*.

c) Each <interface identifier> of a <signal list> of an <interface use list> represents corresponding *Signal-identifier* items in the *Signal-identifier-**set***: one for each *Signal-definition* of *Signal-definition-**set*** of the identified *Interface-definition*.

d) Each *Signal-identifier* in the *Signal-identifier-**set*** appears only once, even if it corresponds to more than one item in the <signal list> of an <interface use list>.

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.

Semantics

The *Sort* of an *Interface-definition* is a pid sort. Each element of the *Sort* is the identity of an agent instance that is able to receive each of the signals identified by the *Signal-identifier-**set*** of the *Interface-definition*.

The *Null-literal-signature* of an *Interface-definition* is the signature for the null literal operator.

The *Data-type-identifier* set of an *Interface-definition* identifies the base interface types (super types) of an interface specialization. `Pid` is directly or indirectly the base interface types of any *Interface-definition* for a pid sort.

NOTE 5 – Specialization is defined in [ITU-T Z.102] and is not included in Basic SDL-2010, but the abstract syntax is included so that *Interface-definition* does not have to be redefined in [ITU-T Z.102].

The *Signal-identifier-**set*** of an *Interface-definition* identifies signals defined outside the interface that are used by the interface and also each signal defined by a *Signal-definition* of the *Interface-definition*.

The *Signal-definition-**set*** of an *Interface-definition* is the set of signals defined for the interface. The scope of the signals is such that they are visible wherever the interface is visible. These signals are included in the *Signal-identifier-**set*** of the *Interface-definition*.

The *Signal-identifier-**set*** of an *Interface-definition* is the set of signal identities that apply when the interface appears in the syntax, and the set of signal identities for the pid sort of the *Interface-definition*. An identity of an agent instance is compatible with the pid sort if every *Signal-identifier* set of the pid sort is in the valid input signal set of the agent.

The implicit *Interface-definition* for an agent type (or agent or state machine) has a *Sort* with the same *Name* as the agent type (or agent or state machine respectively).

Internally connected gates of an agent type are gates of the agent type that are connected via channels to the gates of either a contained agent or the state machine of the agent type. The internally connected gates of an agent are the gates of the agent that correspond to the internally connected gates of the agent type for the agent.

The implicit *Interface-definition* defined by an agent type contains (in its interface specialization – see [ITU-T Z.104]) all interfaces given in the incoming signal lists associated with internally connected gates. The *Interface-definition* contains in its *Signal-identifier-**set*** all signals given in the incoming signal lists associated with internally connected gates.

The implicit *Interface-definition* defined by a state machine of an agent type contains (in its interface specialization – see [ITU-T Z.104]) the interface defined by the agent type itself except any part of that interface concerned only with contained agents. The interface also contains in its interface specialization all interfaces given in the incoming signal lists associated with any gates of the state machine. The interface also contains in its <interface use list> all signals given in the incoming signal lists associated with gates of the state machine.

The *Signal-identifier-**set*** of an implicit *Interface-definition* for an agent type (or agent or state machine) has a *Signal-identifier* for each different signal in the set of signals in all channels or gates for which the destination is the agent type (or agent or state machine respectively), plus the valid input signal set defined explicitly for the agent type (or agent or state machine respectively).

The implicit *Interface-definition* of a typebased agent contains the same *Signal-identifier-**set*** as the *Interface-definition* defined by its type.

NOTE 6 – In Comprehensive SDL-2010 agent definitions given without explicitly defining an agent type represent an agent based on a type given by the body of the agent definition. This model is expanded before interfaces for the agent and its (anonymous) agent type are derived as above.

NOTE 7 – Because every agent and agent type has an implicitly defined interface with the same name, any explicitly defined interface has to have a different name from every agent and agent type defined in the same scope; otherwise, there are name clashes.

12.1.3 Operation signature

Abstract grammar

Static-operation-signature	::	*Operation-signature*
Operation-signature	::	*Operation-name* *Formal-argument** [*Operation-result*] *Procedure-identifier*

Operation-name	=	*Name*
Formal-argument	::	*Argument*
Operation-result	::	*Sort-reference-identifier*
Argument	=	*Sort-reference-identifier*

The *Procedure-identifier* in an operator signature is an anonymous identifier for the anonymous procedure corresponding to the operation.

Concrete grammar

```
<operation signatures> ::=
                [<operator list>]

<operator list> ::=
                operators <operation signature> { <end> <operation signature> }* <end>

<operation signature> ::=
                <operation name>
                        [<arguments>] <result>

<operation name> ::=
                <operation name>
        |       <quoted operation name>
```

NOTE – An <operation name> has to be a <name>, whereas in SDL-2000 it is a name or a number (an <integer name> or a <real name>).

```
<arguments> ::=
                ( <argument> { , <argument> }* )

<argument> ::=
                <formal parameter>

<formal parameter> ::=
                <parameter kind> <sort>

<result> ::=
                <result sign> <sort>
```

An <operation signature> of an <operator list> represents a *Static-operation-signature*.

In an *Operation-signature*, each *Sort-reference-identifier* in *Formal-argument* is represented by an argument <sort>, and the *Operation-result* is represented by the result <sort>. The <sort> in the <formal parameter> of an <argument> of an operation represents the *Formal-argument*.

The *Operation-name* is unique within the defining scope unit in the abstract syntax even though the corresponding <operation name> is not necessarily 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 operation names that have special syntactic forms. The special syntax is introduced so that common operations (for example, arithmetic operations and Boolean operations) have their usual infix syntactic form. That is, the user writes "(1 + 1) **rem** 2" rather than having to use, for example, rem(add(1,1),2).

Semantics

The quoted forms of infix or monadic operations are valid names for operators and each has a corresponding *Name*.

An operation has a result sort, which is the sort identified by the result.

12.1.4 Generic data type operations

Every value data type includes generic operations.

Concrete grammar

The constructor that includes a particular generic operation also represents the *Procedure-definition* for the generic operation in the *Procedure-definition-**set*** of the directly enclosing *Data-type-definition*. The *Procedure-name* of the *Procedure-definition* for the generic operation is an anonymous unique name, and the *Procedure-definition* is associated with the *Operation-signature* by the *Procedure-identifier* in the *Operation-signature*. The *Procedure-formal-parameter* list and *Result* of the *Procedure-definition* is derived from the *Formal-argument* parameter list and *Operation-result* of the *Operation-signature* with arbitrary, anonymous names given to the parameters. The *Procedure-graph* of the *Procedure-definition* of a generic operation is derived from the language semantics, and it is not possible to specify it explicitly.

For a value data type with the <data type name> `S`, there are two items in the *Operation-signature-**set*** items equivalent to including the following explicit <operation signature> definitions in the <operator list> of its <operation signatures>:

```
equal   ( S,S   ) -> Boolean;
copy    ( S     ) -> S;
```

where the parameters and the results correspond to an *Aggregation-kind* of **PART**.

Semantics

Detail on how generic operators behave for each different type constructor is given in the description of the type constructor, together with additional generic operators for the specific type constructor.

The *Operation-signature-**set*** of a *Value-data-type-definition* includes a generic operation that determines the equality between two values of this sort, and a generic operation that takes the value of the sort from the interpretation of an expression and returns that value.

The *Procedure-graph* of the *Procedure-definition* of a generic operation directly provides the semantics defined for the generic operation.

12.1.5 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 are allowed to be assigned to that variable.

Concrete grammar

The interface data type `Pid` represents the unique *Interface-definition* with an empty *Data-type-identifier* set, an empty *Signal-definition* set and an empty *Signal-identifier* set. The *Null-literal-signature* of the `Pid` sort is the unique named element of the `Pid` sort, denoted by `null`. The interface data type `Pid` is optionally qualified by **`package`** `Predefined`.

An *Interface-definition* represented by an <interface definition> (without an interface specialization; see [ITU-T Z.104]) contains only a *Data-type-identifier* denoting the interface `Pid`. The *Null-literal-signature* of a pid sort is a unique named element of the sort, denoted by `null`.

The `Pid` sort and each pid sort has generic *Operation-signature* items equivalent to the following <operation signature> items in the <operator list>:

```
Null        -> P;
Make        -> P;
```

where `P` is the name of the sort (`Pid` or the name of the pid sort).

Semantics

The `Pid` sort is a sort that contains elements that identify agent instances and a unique named element, the *Null-literal-signature* that does not identify any agent. Each agent instance has a corresponding unnamed element in the sort `Pid`. A variable or expression of the `Pid` sort is therefore allowed to identify any agent instance or the *Null-literal-signature* of the `Pid` sort.

For the `Pid` sort, the generic operator `equal` is `true` if its two operands identify the same agent instance or its two operands are the *Null-literal-signature*. For the `Pid` sort, the generic operator `equal` is `false` if its two operands identify different agent instances or only one of its two operands is the *Null-literal-signature*.

For the `Pid` sort, if the operand of the generic operator `copy` is an expression that identifies an agent instance, the generic operator `copy` returns the identity of that agent instance; otherwise it returns the *Null-literal-signature* for the `Pid` sort.

The operator `Null` of the `Pid` sort returns the *Null-literal-signature* of the `Pid` sort. The operator `Make` of the `Pid` sort creates a new instance of the `Pid` sort associated with the *Null-literal-signature* of the `Pid` sort. An attempt to obtain an associated agent identity from the *Null-literal-signature* of the `Pid` sort raises the predefined exception `InvalidReference`.

A pid sort is a *Sort* introduced by an *Interface-definition*. All the signals of a pid sort are all the signals in the *Signal-identifier-**set*** of the *Interface-definition* that introduces the pid sort.

For a pid sort, the generic operator `equal` is `true` if its two operands identify the same agent instance or its two operands are the *Null-literal-signature*. For a pid sort, the generic operator `equal` is `false` if its two operands identify different agent instances or only one of its two operands is the *Null-literal-signature*.

For a pid sort, if the operand is an expression that identifies an agent instance, the generic operator `copy` returns the identity of the agent instance identified by the operand if that agent instance that accepts all the signals of the pid sort, otherwise it returns the *Null-literal-signature* for the pid sort.

The operator `Null` of a pid sort returns the *Null-literal-signature* of the pid sort. An attempt to obtain an associated agent identity from the *Null-literal-signature* of the pid sort raises the predefined exception `InvalidReference`. The operator `Make` of a pid sort creates a new instance of the pid sort associated with the *Null-literal-signature* of the pid sort.

Each pid sort is based (directly or indirectly through another pid sort) on the `Pid` sort and therefore contains a named element for the *Null-literal-signature* of the `Pid` data type that does not identify with any agent. As well as the named element, a pid sort contains elements that identify the agent instances that accept all the signals of the pid sort. A variable or expression of the pid sort is only allowed to identify agent instances that accept all the signals of the pid sort. For example, a variable with the pid sort introduced by the *Interface-definition* defined by an agent definition is able to be associated with the identity of the agent from the interpretation of a *Create-request-node* (see clause 11.13.2) for the agent.

NOTE 1 – A pid sort expression is allowed to identify an agent that accepts additional signals provided the agent accepts all the signals of the pid sort.

Each interface adds an output compatibility check operation that, given a signal and a destination with a pid sort, determines whether either:

a) the signal is defined or used in the interface of the destination: that is, the *Signal-identifier* of the signal being output is in the *Signal-identifier-**set*** of the *Interface-definition* for the destination sort; or

b) the output compatibility check is satisfied for a pid sort introduced by an *Interface-definition* identified by the *Data-type-identifier-**set*** (that is, the interface specialization) of the destination sort (see [ITU-T Z.104]).

NOTE 2 – In Basic SDL-2010 there is no specialization of interfaces, so the check in b) does not apply.

If the output compatibility check operation is not fulfilled, the predefined exception `InvalidReference` is raised.

12.1.6 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 obtained by constructing a tuple from elements of given sorts.

Concrete grammar

<data type constructor> ::=
  <literal list>
  | <structure definition>
  | <choice definition>

12.1.6.1 Literals constructor

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-natural*
Literal-natural	=	*Nat*
Literal-name	=	*Name*

Each *Literal-signature* in the *Literal-signature* set of a *Value-data-type-definition* shall have a different *Literal-natural*.

Concrete grammar

<literal list> ::=
  literals <literal signature> { **,** <literal signature> }* <end>

<literal signature> ::=
  <literal name>
  | <named number>

<literal name> ::=
  <<u>literal</u> name or number>

<named number> ::=
  <literal name> <equals sign> <<u>Natural</u> 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 if the corresponding <literal name> is not 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.

The <u>Natural</u> simple expression value of the <<u>Natural</u> simple expression> occurring in a <named number> represents the *Literal-natural* of the *Literal-signature.*

Each <literal name> in a <literal list> is given the lowest possible Natural simple expression value for the *Literal-natural* of the *Literal-signature* not occurring for any other <literal signature>s of the same <literal list>, considering the <literal name>s one by one from left to right. The result is, for example,

```
        literals B, A = 2, C, D;
```

has B < C , C < A , A < D , num(C) = 1, num(D) = 3

Semantics

Each element in the sort is represented by a *Literal-signature* and has a *Literal-natural* that has a corresponding Natural simple expression value.

The result of the generic operator `equal` is `true` if and only if its two operands represent the same *Literal-signature* (that is, they represent the same element of the sort). The result of the generic operator `copy` is the same as the actual argument value.

Additional generic operators exist for a sort defined by a constructor that creates a *Literal-signature* set, as follows:

a) an operator that gives the position of each data item in the ordering as the corresponding Natural simple expression value;

b) operators that compare two data items with respect to the established ordering; and

c) operators that return the first, last, next or previous data item in the ordering.

For a sort named S that is defined by a constructor that creates a *Literal-signature* set, there is a *Static-operation-signature* list equivalent to the following:

```
        num  ( S )    -> Natural;
        "<"  ( S, S ) -> Boolean;
        ">"  ( S, S ) -> Boolean;
        "<=" ( S, S ) -> Boolean;
        ">=" ( S, S ) -> Boolean;
        first         -> S;
        last          -> S;
        succ ( S )    -> S;
        pred ( S )    -> S;
```

where Boolean is the predefined Boolean sort and Natural is the predefined Natural sort, and the parameters and the results correspond to an *Aggregation-kind* of **PART**.

The operator `num` returns the Natural simple expression value corresponding to the *Literal-natural* of the literal.

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 values corresponding to each *Literal-natural* of the two literals. The operator `first` returns the first data item in the ordering (the literal with the lowest Natural simple expression value corresponding to the *Literal-natural*). The operator `last` returns the last data item in the ordering (the literal with the highest Natural simple expression value corresponding to the *Literal-natural*). The operator `pred` returns the preceding data item (that is, the literal with the highest *Literal-natural* that is less than the *Literal-natural* corresponding to the actual parameter), if one exists, or the same as the operator `last`, otherwise. The operator `succ` returns the successor data item (that is, the literal with the lowest *Literal-natural* that is greater than the *Literal-natural* corresponding to the actual parameter) in the ordering, if one exists, or the same as the operator `first`, otherwise.

12.1.6.2 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 grammar

```
<structure definition> ::=
        struct [<field list>] <end>

<field list> ::=
        <field> { <end> <field> }*

<field> ::=
    |   <optional field>
    |   <mandatory field>

<optional field> ::=
        <fields of sort> optional

<mandatory field> ::=
        <fields of sort> [ <field default initialization> ]

<fields of sort> ::=
        <field of kind> <field sort>

<field of kind> ::=
        <aggregation kind> <field name>

<field default initialization> ::=
        default <constant expression>

<field sort> ::=
        <sort>
```

NOTE 1 – The <field name> of a structure sort has to be a <name>, whereas in SDL-2000 it is a name or a number (an <integer name> or a <real name>).

Each <field name> of a structure sort shall be different from every other <field name> of the same <structure definition>.

The <structure definition> for a structure `S` represents (in the *Operation-signature* set of the *Data-type-definition* for `S`):

a) in the absence of data type specialization (see [ITU-T Z.104] clause 12.1.9), if no operator named `Make` is given with an *Operation-result* that is the *Sort-reference-identifier* of the `S` structure sort (an `S` structure result), an *Operation-signature* for a generic operator named `Make` with:

   i. a *Formal-argument* list where each item is the *Sort-reference-identifier* of the corresponding (in order) <field name> if the referenced <field> does not contain **optional** and does not contain a <field default initialization>;,

   ii. an *Operation-result* that is the `S` structure result;

   iii. the procedure identified by the *Operation-signature* having each formal parameter of its *Parameter-aggregation* derived from the <aggregation kind> of the corresponding <field name>, and a *Result-aggregation* that is **PART**.

b) if data type specialization is present, [ITU-T Z.104] applies and the *Operation-signature* for a generic operator named `Make` is determined as described in [ITU-T Z.104] clause 12.1.6.2.

c) for each field, if the <field name> is `fn` and the <field sort> is `fs`, an *Operation-signature* for the <operation signature>
   ```
   fnExtract ( S ) -> fs;
   ```
   for a generic operator where
   `fnExtract` is a *field-extract-name* formed from the concatenation of the field name and "Extract",
   and in the procedure identified by the *Operation-signature*
   `S` is an **in/out** parameter with **PART** *Parameter-aggregation* for `S`
   and the *Result-aggregation* is derived from the <aggregation kind> field `fn`.

NOTE 3 – A special syntax is provided as described in clause 12.2.3. To use `fnExtract` to extract the value of field `fn` from a structure variable `vs` and assign the value to `Variable` (a variable with the sort of field `fn`), the notation is:

```
Variable := vs.fn;
```

d) for each field, if the <field name> is `fn` and the <field sort> is `fs`, an *Operation-signature* for the <operation signature>

```
fnModify ( S, fs ) -> S;
```

for a generic operator where

`fnModify` is a *field-modify-name* formed from the concatenation of the field name and "Modify",

and in the procedure identified by the *Operation-signature*

`S` is an **in/out** parameter with **PART** *Parameter-aggregation*,

`fs` is an **in** parameter with *Parameter-aggregation* derived from the <aggregation kind> of field `fn`,

and the *Result-aggregation* is **PART**.

NOTE 4 – A special syntax is provided as described in clause 12.3.3.1. To use `fnModify` to assign the value of `Variable` (a variable with the sort of field `fn`) to field `fn` of a structure variable `vs`, the notation is:

```
vs.fn := FieldValue;
```

e) for each field, if the <field name> is `fn`, an *Operation-signature* for the <operation signature>

```
fnPresent ( S ) -> <<package Predefined>>Boolean;
```

for a generic operator where

`fnPresent` is a *field-present-name* formed from the concatenation of the field name and "`Present`",

and in the procedure identified by the *Operation-signature*

`S` is an **in/out** parameter with **PART** *Parameter-aggregation*,

and the *Result-aggregation* is **PART**.

NOTE 5 – In SDL-2000 `fnPresent` is only defined if the field is optional or has a default value.

f) an *Operation-signature* for a generic operator named `Undefined` based on the <operation signature>

```
Undefined ( S )-> <<package Predefined>>Boolean;
```

which is `true` if the structure is "undefined": that is, every field of the structure is "undefined",

and in the procedure identified by the *Operation-signature*

`S` is an **in/out** parameter with **PART** *Parameter-aggregation*,

and the *Result-aggregation* is **PART**.

Semantics

A structure sort has elements that are all the tuples constructed from data items belonging to the sorts given in the field list. An element of a structure sort has as many component elements as there are fields in the field list. An optional field is a field that does not have to be present. The associated operations determine the semantics of the structures sort.

The result of the generic operator `equal` is `true` if and only if for each field of the structure sort:

a) the field is not present in both operands of `equal`; or

b) the field is present in both operands of `equal`, and `equal` for the sort of the field between the values of the field in two structures is `true`.

The generic operator `copy` behaves as if the following is interpreted:

a) a new structure is created in which each field has no value (it is "undefined"); then

b) for each field that is present in the operand of `copy`, the corresponding field of the structure is associated with the data item associated with that field in the operand of `copy`;

c) for each field that is not present in the operand of `copy`, and the corresponding field of the structure has a default initialization, the field of the structure is associated with the data item for that field in the default initialization.

Additional generic operations exist for the sort defined as a structure as follows:

a) operations to create structure values;

b) operations to modify structures and to access component data items of structures values; and

c) an operation to test for the presence of optional component data items in structures values, or if the structure is "undefined".

A `Make` operation with an empty *Formal-argument* list creates a structure value with values associated with fields that have default initialization and all other fields "undefined".

A `Make` operation with a non-empty *Formal-argument* list creates a new structure and associates each field with the result of the corresponding formal parameter, or if no actual argument is given for the field, the default initialization for that field, or "undefined" if there is no default initialization for the field.

If, during interpretation, a field of a structure is "undefined", applying the operation to access this field (with a *field-extract-name*) to the structure causes the predefined exception `UndefinedField` to be raised. Otherwise, the operation to access a field returns the data item associated with that field. The value associated with a structure is not changed by interpretation of the operation to access a field of the structure.

The operation to modify a field (with a *field-modify-name*) associates the field with the result of its argument *Expression*. The value associated with the structure after interpreting the operation has the field associated with the argument value, but no change to the value associated with any other field.

The operation to test for the presence of a field data item based on the field name (with a *field-present-name*) returns the predefined `Boolean` value `false` if this field is "undefined", and the predefined `Boolean` value `true` otherwise. The value associated with a structure is not changed by interpretation of the operation to test presence of a field of the structure.

The `Undefined` operation tests if the structure is "undefined", and returns the `Boolean` value `true` if each of the fields of the structure is "undefined". The value associated with a structure is not changed by interpretation of the operation to test if the structure is "undefined".

12.1.6.3 Choice data types

A choice data type constructor is a notation for defining data type similar to a structure type with all fields optional, and that every data item always has at most one component field present or no fields present at all.

Concrete grammar

<choice definition> ::=

choice [<choice list>] <end>

<choice list> ::=

<choice of sort> { <end> <choice of sort> }*

<choice of sort> ::=

<aggregation kind> <<u>field</u> name> <field sort>

NOTE 1 – The <<u>field</u> name> of a choice sort has to be a <name>, whereas in SDL-2000 it is a name or a number (an <integer name> or a <real name>).

Each <<u>field</u> name> of a choice sort shall be different from every other <<u>field</u> name> of the same <choice definition>.

The <choice definition> for a choice sort `C` represents (in the *Operation-signature* set of the *Data-type-definition* for `C`):

a) an *Operation-signature* for a generic operator named `Make` with an empty *Formal-argument* list and an *Operation-result* that is the *Sort-reference-identifier* of the `C` choice sort, and the procedure identified by the *Operation-signature* has a *Result-aggregation* that is **PART**.

b) for each field, if the <field name> is `fn` and the <field sort> is `fs`, an *Operation-signature* for the operation signature
```
fn ( fs ) -> C;
```
for a generic field association operator where
`fn` is a *field-associate-name* which is the same as the field name,
and in the procedure identified by the *Operation-signature*
`fs` is an **in** parameter with *Parameter-aggregation* derived from the <aggregation kind> of field `fn`,
and *Result-aggregation* is **PART**.

c) for each field, if the <field name> is `fn` and the <field sort> is `fs`, an *Operation-signature* for the <operation signature>
```
fnExtract ( C ) -> fs;
```
for a generic operator where
`fnExtract` is a *field-extract-name* formed from the concatenation of the field name and "Extract",
and in the procedure identified by the *Operation-signature*
`C` is an **in/out** parameter with **PART** *Parameter-aggregation*,
and *Result-aggregation* is derived from the <aggregation kind> of field `fn`.

NOTE 2 – A special syntax is provided as described in clause 12.2.3. To use `fnExtract` to extract the value of field `fn` from a choice variable `vc` and assign the value to `Variable` (a variable with the sort of field `fn`), the notation is:
```
Variable := vc.fn;
```

d) for each field, if the <field name> is `fn` and the <field sort> is `fs`, an *Operation-signature* for the <operation signature>
```
fnModify (C, fs ) -> C;
```
for a generic operator where
`fnModify` is a *field-modify-name* formed from the concatenation of the field name and "Modify",
and in the procedure identified by the *Operation-signature*
`C` is an **in/out** parameter with **PART** *Parameter-aggregation*,
`fs` is an **in** parameter with *Parameter-aggregation* derived from the <aggregation kind> of field `fn` ,
and *Result-aggregation* is **PART**.

NOTE 3 – A special syntax is provided as described in clause 12.3.3.1. To use `fnModify` to assign the value of `Variable` (a variable with the sort of field `fn`) to field `fn` of a choice variable `vc`, the notation is:
```
vc.fn := FieldValue;
```

e) for each field, if the <field name> is `fn`, an Operation-signature for the <operation signature>
```
fnPresent ( C ) -> <<package Predefined>>Boolean;
```
for a generic operator where
`fnPresent` is a *field-present-name* formed from the concatenation of the field name and "`Present`",
and in the procedure identified by the *Operation-signature*
`C` is an **in/out** parameter with **PART** *Parameter-aggregation*,
and *Result-aggregation* is **PART**.

f) an *Operation-signature* for a generic operator named `PresentExtract` based on the <operation signature>
`PresentExtract ( C )-> AnonPresent;`
where `AnonPresent` is defined as a literal constructor data type that uses the field names of the choice as literals as described below,
and in the procedure identified by the *Operation-signature*
`C` is an **in/out** parameter with **PART** *Parameter-aggregation*,
and *Result-aggregation* is **PART**.

g) an *Operation-signature* for a generic operator named `Undefined` based on the <operation signature>
`Undefined (C )-> <<`**`package`**` Predefined>>Boolean;`
which is `true` if the choice is "undefined",
and in the procedure identified by the *Operation-signature*
where `C` is an **in/out** parameter with **PART** *Parameter-aggregation*,
and *Result-aggregation* is **PART**.

The <choice definition> for a choice sort `C` also represents an additional (anonymous) *Data-type-definition*, that for description above is called `AnonPresent`. This *Data-type-definition* is placed in the *Data-type-definition* for `C`, therefore both `AnonPresent` and its contained *Literal-signature-**set*** are visible where `C` is visible. This is defined with a *Literal-signature-**set*** where each <field name> of the choice sort `C` represents a *Literal-signature*. The order of the literals is the same as the order in which the <field name>s are specified left to right in the choice sort `C`. The purpose of this data type is to allow the operation `PresentExtract` with a result that corresponds to the field name. The name of this data type being unknown prevents it being used for other purposes.

Semantics

A choice sort has elements that contain information about the current field that the choice holds, and the value of that field. If any field of the choice is assigned a value, the previous value is lost regardless of which field it was in.

The result of the generic operator `equal` is `true` if and only if:

a) the field present in both operands of `equal` is the same field; and

b) `equal` for the sort of the field between the values of the field in the two choices is `true`.

NOTE 4 – If either operand of `equal` is "undefined", the predefined exception `UndefinedField` is raised when the operand is accessed.

The generic operator `copy` behaves as if the following is interpreted:

a) a new choice is created (each field has no value – it is "undefined"); then

b) for the field that is present in the operand of `copy`, the corresponding field of the choice is associated with the data item associated with that field in the operand of `copy`.

NOTE 5 – If the operand of `copy` is "undefined" – that is, every field is "undefined" – the predefined exception `UndefinedField` is raised when the operand is accessed.

Additional generic operations exist for the sort defined as a choice as follows:

a) operations to create a choice;

b) operations to modify a choice and to access the current choice field; and

c) operations to test for the presence of a particular choice field, or if the choice is "undefined", which field is present.

The `Make` operation for a choice creates an "undefined" choice value, which is assignable to a choice variable. Every field of such a choice value is "undefined". Applying an operation to access any field of an "undefined" choice value or to determine which field is present in an "undefined" choice value raises the predefined exception `UndefinedField`.

The generic field association operator (with a *field-associate-name*) creates a choice value where the field with the same name is associated with the result of its argument *Expression*. The result is assignable to a choice variable. Every other field of such a choice value is "undefined". Applying an operation to access any "undefined" field of the choice value raises the predefined exception `UndefinedField`.

NOTE 6 – Applying the generic field association operator for the choice has the same result as applying the operation to modify the specified field with the given field value *Expression* to the result of `Make` operation for the choice.

If, during interpretation, the field of a choice is "undefined", applying the operation to access this field (with a *field-extract-name*) to the choice causes the predefined exception `UndefinedField` to be raised. Otherwise, the operation to access a field returns the data item associated with that field. The value associated with a choice is not changed by interpretation of the operation to access a field of the choice.

The operation to modify a field (with a *field-modify-name*) associates the field with the result of its argument *Expression*. The choice value after interpreting the operation is not "undefined" and has the field associated with the argument value, and all other fields are "undefined".

The operation to test for the presence of a field data item based on the field name (with a *field-present-name*) returns the predefined Boolean value `false` if this field is "undefined", and the predefined Boolean value `true` otherwise. The value associated with a choice is not changed by interpretation of the operation to test presence of a field of the choice.

The `Undefined` operation tests if the structure is "undefined", and returns the `Boolean` value `true` if each of the fields of the structure is "undefined". The value associated with a structure is not changed by interpretation of the `Undefined` operation.

If, during interpretation, the field of a choice is "undefined", testing which field is present by the application of the `PresentExtract` operation to the choice causes the predefined exception `UndefinedField` to be raised. Otherwise, the `PresentExtract` operation returns the value of the `AnonPresent` sort with the same name as the field that is present. The value associated with a structure is not changed by interpretation of the `PresentExtract` operation.

12.1.7 Behaviour of operations

A <data type definition> allows operations to be added to a data type. The behaviour of operations is defined in a manner similar to value returning procedure calls. The operations of a data type never access or change the global state of the input queues of the agents in which they are called, therefore only contain a single transition. Otherwise an operation behaves as a restricted kind of procedure and the *Concrete grammar* represents *Abstract grammar* and *Semantics* that are described in clause 9.4.

Concrete grammar

<operation definitions> ::=
        <operation definition item>*

<operation definition item> ::=
        <operation reference>

<operation heading> ::=
        operator [<qualifier>] <operation name>
            [<formal operation parameters>]
            [<operation result>]

If the <operation name> of the <operation heading> is a <quoted operation name> that is <quotation mark> <equals sign> <quotation mark> or is <quotation mark> <not equals sign> <quotation mark>, the number of <formal operation parameters> shall be less than one or greater than two.

NOTE 1 – It is not allowed to define an operation with two parameters and an <operation name> that is <quotation mark> <equals sign> <quotation mark> or <quotation mark> <not equals sign> <quotation mark>. This avoids any ambiguity between an *Operation-application* and an *Equality-expression* for the generic equal operator.

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

<formal operation parameters> ::=
                    [ <end> ] fpar <formal variable parameters> {, <formal variable  parameters> }*
```

NOTE 2 – The syntax of SDL-2000 that uses round brackets rather than **fpar** is in [ITU-T Z.104].

The list of <<u>variable</u> name>s of the <parameters of sort> of the <formal variable parameters> is considered to bind tighter than the list of <formal variable parameters> within <formal operation parameters>.

```
<operation result> ::=
                    returns <result aggregation> <sort>

<operation diagram> ::=
                    <operation page>

<operation page> ::=
                    <frame symbol> contains {
                            <operation heading> <page number area>
                            { <operation text area>* <operation body area> }set }
                    [ is associated with <package use area> ]

<operation body area> ::=
                    { [ <procedure start area> ]
                            { <in connector area> }* } set
```

NOTE 3 – The <procedure start area> of an <operation body area> is optional only if the diagram has multiple pages (see [ITU-T Z.104]).

```
<operation text area> ::=
                    <text symbol> contains
                    {       <data definition>
                    |       <variable definition> }*
```

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

For each <operation signature> at most one corresponding <operation diagram> is given.

<operation body area> shall contain neither an <imperative expression> nor an <identifier> defined outside the enclosing <operation diagram>, except for <operation identifier>s, <literal identifier>s and <sort>s.

In Basic SDL-2010 the <operation diagram> shall have a corresponding <operation signature> in the <operations> of the data type, which represents the corresponding *Operation-signature*.

An <operation diagram> represents a *Procedure-definition* in the *Procedure-definition-**set*** of the directly enclosing *Data-type-definition*. The *Procedure-name* of the *Procedure-definition* is an anonymous unique name, and the *Procedure-definition* is associated with the *Operation-signature* by the *Procedure-identifier* in the *Operation-signature*. The <formal operation parameters> list for the operator represents the *Procedure-formal-parameter* list of the *Procedure-definition* in the same way as the formal parameters for a procedure. The <operation result> represents the *Result* of the *Procedure-definition*, therefore the <result aggregation> represents the *Result-aggregation*. The components of the <operation body area> are used in the same way as the components of a

<procedure body area> to represent the *Data-type-definition-**set***, *Syntype-definition-**set***, *Variable-definition-**set***, *Procedure-definition-**set*** and *Procedure-graph* of the *Procedure-definition*.

Semantics

An operator is a procedure with an anonymous unique *Procedure-name* and a *Procedure-definition* in a directly enclosing *Data-type-definition*, where the *Procedure-definition* is associated with the *Operation-signature* for the operator by the *Procedure-identifier* in the *Operation-signature* for the operator in the *Data-type-definition*.

An operator is a constructor for elements of the sort identified by the result. Usually the same element of the sort is constructed by several operators and there is no unique set of constructor operators for a sort, though literal operators are usually considered the prime constructor for the literal they represent: for example, `true` and `false` for `Boolean`.

An operator shall not modify actual parameters or any items defined outside the scope of the operator.

An operation is a scope unit defining its own data and variables that are allowed to be manipulated inside the operation.

Variables introduced in formal parameters (that is, the corresponding *Procedure-formal-parameter* list) are local variables of the operation, and modification within the operation is allowed.

If an operation contains informal text, the interpretation of expressions involving the application of the corresponding operation is not formally defined by the Specification and Description Language but is determined from the informal text by the interpreter. If informal text is specified, a complete formal specification has not been given in the Specification and Description Language.

12.1.8 Additional data definition constructs

This subclause introduces further constructs for data.

12.1.8.1 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* [*Default-initialization*]
Syntype-name	=	*Name*
Parent-sort-identifier	=	*Sort-identifier*

Concrete grammar

<syntype> ::=
<syntype identifier>

<syntype definition> ::=
{<package use clause>}* { <syntype definition syntype> }

<syntype definition syntype> ::=
syntype <syntype name> <equals sign> <parent sort identifier>
{ [<comment body>] <left curly bracket>
[{ <default initialization> [[<end>] <constraint>] | <constraint> } <end>]
<right curly bracket>
}

A <comment body> is a form of annotation and has no formal semantic meaning.

NOTE 1 – Each <syntype name> has to be a <name>, whereas in SDL-2000 it is a name or a number (an <integer name> or a <real name>).

<parent sort identifier> ::=
    <sort>

A <syntype> is an alternative for a <sort>.

A <syntype definition> with <syntype definition syntype> 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* is the *Parent-sort-identifier* of the syntype.

When a <syntype identifier> is used as a result of an operation, the sort of the *Operation-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 <constraint> is omitted, the <syntype identifier>s for the syntype are in the Abstract grammar represented as the *Parent-sort-identifier*.

If a <constraint> could be interpreted as either belonging to the <default initialization> or the <syntype definition>, it shall be considered part of the <default initialization>.

When a syntype is specified in terms of <syntype identifier>, the two syntypes shall not be mutually defined: that is, the <parent sort identifier> of a <syntype definition> shall not refer directly or indirectly to the identity of the syntype being defined.

Semantics

A syntype definition defines a syntype, which identifies a sort identifier (*Parent-sort-identifier*) and has a constraint (*Range-condition*). Specifying a syntype identifier is the same as specifying the parent sort identifier of the syntype, except for the following cases:

d) assignment to a variable declared with a syntype (see clause 12.3.3);

e) output of a signal if one of the sorts specified for the signal is a syntype (see clause 10.3 and clause 11.13.4);

f) calling a procedure when one of the sorts specified for the procedure in parameter variables is a syntype (see clause 9.4 and clause 11.13.3);

g) creating an agent when one of the sorts specified for the agent parameters is a syntype (see clauses 9.2, 9.3 and 11.13.2);

h) input of a signal and one of the variables associated with the input has a sort that is a syntype (see a));

i) calling an operation application that has a syntype defined as either an argument sort or a result sort (see clause 12.2.6);

j) set or reset clause or active expression on a timer and one of the sorts in the timer definition is a syntype (see clause 11.15 and clause 12.3.4.3);

k) procedure formal context parameter with an in/out or out parameter in a 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 (see [ITU-T Z.102]; context parameters are not allowd in Basic SDL-2010).

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.

12.1.8.2 Constraint

Abstract grammar

Range-condition	::	*Condition-item-**set***
Condition-item	=	*Open-range* | *Closed-range* | *Size-constraint*
Open-range	::	*Operation-identifier* *Constant-expression*
Closed-range	::	*Constant-expression* *Constant-expression*
Size-constraint	::	*Operation-identifier* { *Open-range* | *Closed-range* }*

Concrete grammar

<constraint> ::=
    constants (<range condition> **)**
  | <size constraint>

<range condition> ::=
    <range> { **,** <range> }*

<range> ::=
    <closed range>
  | <open range>

<closed range> ::=
    <constant> { <colon> | <range sign> } <constant>

<open range> ::=
    <constant>
  | <open range with operator>

<open range with operator> ::=
    {   <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 syntype of a <range condition> in a <constraint> of a data type definition (including inline definitions and <basic sort> with a <range condition>) is the syntype defined by the data type definition. The syntype for a <range condition> in a <size constraint> is `Natural`. The syntype or data type of a <range condition> in an <answer> is the syntype or data type of the corresponding <question>.

The symbol "<" shall 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 or data type for the context of the <range condition>, and similarly for the symbols (`"<="`, `">"`, `">="`, respectively). These symbols represent *Operation-identifier*.

A <closed range> shall 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 or data type for the context of the <range condition>.

A <constant expression> in a <range condition> shall have the same sort as the sort of the syntype.

A <size constraint> shall only be used in the concrete syntax of a <constraint> if `length` has been defined as an operation with the <operation signature>:

```
length (in P ) -> <<package Predefined>>Integer;
```

where `P` is the sort of the syntype for the context of the <constraint>.

A <constraint> defines a range check: that is, the *Range-condition* operation to be applied. The range check is derived as follows where `constant`, and `secondconstant` are <constant> items, and `RC` is a <range condition> item:

a) Each <open range> or <closed range> or <size constraint> in the <constraint> has a corresponding *Open-range* or *Closed-range* or *Size-constraint* in the *Condition-item-**set***.

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`, `<= constant`, `> constant`, and `>= constant` has sub-expression in the range check of the form `A = constant`, `A /= constant`, `A < constant`, `A <= constant`, `A > constant`, and `A >= constant`, respectively;

  2) <closed range> of the form `constant : secondconstant` has a sub-expression in the range check of the form `((constant <= A) and (A <= secondconstant))` where **"and"** is the predefined Boolean "`and`" operator;

  3) a <size constraint> of the form **`size`** `(RC)` has a corresponding *Size-constraint* in the *Condition-item-**set*** where the *Operation-identifier* identifies an implicit operator with an anonymous unique name in the sort `P` of the syntype for the range check, with a formal parameter `A` of sort `P`, a `Boolean` result and a body which is a value return that is a distributed "or" (see (d) below) over sub-expressions of the form:

    i) `length(A) = OP`, for each <open range> `OP` in `RC` of the form `constant`;

    ii) `length(A) OP`, for each <open range> `OP` in `RC` of the form `= constant`, `/= constant`, `< constant`, `<= constant`, `> constant`, and `>= constant`;

    iii) `((constant <= length(A)) and (length(A) <= secondconstant ))` where **"and"** is the predefined Boolean "and" operator, for each <closed range> in `RC` of the form `constant : secondconstant`.

d) The predefined Boolean "or" operation is used as a distributed operation over sub-expressions by inserting **`or`** between sub-expressions (that is, for expressions `A B C`…, a distributed operation of the form `A` **`or`** `B` **`or`** `C` …). The range check is the expression formed from this distributed operation over all the sub-expressions corresponding to the *Condition-item-**set***.

If a syntype is specified without a <constraint> then the range check is the predefined Boolean value `true`.

Semantics

A range check is used when a syntype has additional semantics to the sort of the syntype (see clause 12.3.1, clause 12.1.8.1 and the cases where syntypes have different semantics – see the subclauses referenced in items a) to h) under *Semantics* in clause 12.1.8.1). A range check is also used to determine the interpretation of a decision (see clause 11.13.5).

The range check is the application of the operation formed from the *Range-condition*.

For syntype range checks, the application of this operation shall be equivalent to the predefined Boolean value true; otherwise, the predefined exception `OutOfRange` is raised.

12.2 Use of data

This clause defines the general grammar for expressions and how sorts, literals and operators are interpreted in expressions. The use of active expressions that depend on variables and dynamic interpretation is defined in clause 12.3.

12.2.1 Expressions and expressions as actual parameters

The interpretation of an expression gives a value. If this value does not depend on the variables or other active interpretation (such as imperative expressions) the result is a constant value, and the expression is considered to be "passive". Some kinds of expression (such as a literal for a value) are always passive. Other kinds (such as a variable access) always depend on interpretation and are considered "active". Several kinds (such as the application of an operation) have other expressions as elements and are only passive if all the elements are passive, and therefore have a common concrete grammar. For that reason the abstract grammar for both passive *Constant-expression* and *Active-expression* is presented here with the common concrete grammar.

A simple expression is a special kind of constant expression that only uses sorts of data defined in the `Predefined` package.

The abstract grammar of the language has several places where a list of expressions is required as actual parameters for other constructs. The general grammar for these actual parameters is placed here because they are lists of expressions, but additional grammar is applied in the different contexts where the lists are used.

The non-canonical concrete syntax form is given below to avoid extending the <expression> syntax in [ITU-T Z.104] to incorporate the very commonly used familiar infix and prefix forms. The canonical form of the <expression> syntax removes the application of infix and prefix operators such as **and**, **or**, **rem**, **not**, <plus sign> and <concatenation sign> and replaces these with the equivalent <operation application>. For example: `A` **and** `B` is replaced by `"and"(A,B)`; `I + J` is replaced by `"+"(I,J)`.

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*

Actual-parameters :: { *Expression* | UNDEFINED }*

The length of the list of *Expression* and UNDEFINED elements in *Actual-parameters* shall match the number of elements required in the context *Actual-parameters*, if used. In general there is a corresponding list of formal parameters that determines the number of required elements, and each element that is an *Expression* in *Actual-parameters* shall be compatible with the sort of the corresponding by position formal parameter.

Concrete grammar

For simplicity of description, no distinction is made between the concrete syntax of *Constant-expression* and *Active-expression*.

<expression> ::=
<expression0>
| <range check expression>

<expression0> ::=
<operand>
| <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>
| <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>
| <extended primary>
| <active primary>

<active primary> ::=
<variable access>
| <imperative expression>

<expression list> ::=
<expression> { **,** <expression> }*

<simple expression> ::=

<constant expression>

A <simple expression> shall contain only literals, and operations defined within the package `Predefined`, as defined in [ITU-T Z.104].

<constant expression> ::=

<<u>constant</u> expression0>

<actual parameters> ::=

(<actual parameter list> **)**

<actual parameter list> ::=

[<actual parameter>] { **,** [<actual parameter>] }*

<actual parameter> ::=

<expression>

If an <actual parameter> is omitted in the <actual parameter list> of <actual parameters>, this represents UNDEFINED in the abstract syntax, otherwise the <expression> of the <actual parameter> represents the *Expression* in the corresponding position in the *Actual-parameters* list. If the list of items in the <actual parameter list> of <actual parameters> is shorter than the list required in the abstract syntax, this represents additional UNDEFINED elements in the abstract syntax to make the list the correct length. Therefore, it is allowed to omit trailing commas in the <actual parameter list> of <actual parameters>. The <actual parameter list> of <actual parameters> shall not be longer than the list required in the abstract syntax.

An <expression0> that does not contain any <active primary>, or a <value returning procedure call> is a <<u>constant</u> expression0>. A <<u>constant</u> expression0> represents a *Constant-expression* in the abstract syntax.

An <expression> that is not a <<u>constant</u> 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 clause 12.2.3 before relationship to the abstract syntax is considered.

<operand>, <operand1>, <operand2>, <operand3>, <operand4> and <operand5> offer special syntactic forms for operation names. The special syntax is introduced, for example, so that arithmetic operations and Boolean operations have their usual syntactic form. That is, the user writes "(1 + 1) **rem** 2" rather than being forced to use, for example, rem(add(1,1),2). Which sorts are valid for each operation will depend on the data type definition.

An <infix operation name> in an expression has the normal semantics of an operation but with infix or quoted prefix syntax.

A <monadic operation name> in an expression has the normal semantics of an operation but with the prefix or quoted prefix syntax.

The order of precedence of <infix operation name> items determines the binding of operations. When the binding is ambiguous, then binding is from left to right.

Semantics

When an expression is interpreted, it returns a data item (a value 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.

NOTE 1 – In this Recommendation, [ITU-T Z.102], [ITU-T Z.103] and [ITU-T Z.104] only static sorts occur, but they are written so that the language they define is extensible for object-oriented data where polymorphism might occur. Therefore to avoid cumbersome text, the word "sort" always refers to a static sort, but for clarity, "static sort" is written explicitly in some cases where it is anticipated the interpretation might differ if the expression is polymorphic.

Expressions have an aggregation kind, which is propagated from the leaf nodes of an expression tree. Unless otherwise indicated, the aggregation kind of an expression is the aggregation kind of the data item returned by the interpretation of the expression.

NOTE 2 – In this Recommendation, [ITU-T Z.102], [ITU-T Z.103] and [ITU-T Z.104] only PART aggregation kind occurs.

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 replicate of that computed value is used.

When an *Actual-parameters* list is interpreted, each *Expression* is interpreted and is assigned to the formal parameter before the next *Expression* to the right is interpreted. If the sort of the formal parameter is a syntype, the range check defined in clause 12.1.8.2 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` is raised instead of interpreting further actual parameters or further interpretation where the *Actual-parameters* list is used. For each **UNDEFINED** element in the *Actual-parameters* list, the corresponding formal parameter has no data associated with it: that is, it is "undefined".

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* identifies a *Literal-signature*.

Concrete 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 clause 6.2) 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. If there are two literals with the same <name> but differing by result sorts, each has a different *Literal-name*.

It shall be possible to bind each unqualified <literal identifier> to exactly one sort that satisfies the conditions in the construct in which the <literal identifier> is used.

Wherever a <qualifier> of a <literal identifier> ends with a <path item> with the keyword **type**, then the <sort name> is used as the result sort to derive the unique *Name* of the *Identifier*. The *Qualifier* is formed in the usual way from <qualifier>, therefore the <sort name> after the keyword **type** is used for both the *Qualifier* and deriving the *Name* of the *Identifier*.

Semantics

A *Literal* returns the unique data item corresponding to its *Literal-signature*.

A *Literal* has an aggregation kind, which is always **PART.**

The sort of the *Literal* is the *Result* in its *Literal-signature*.

12.2.3 Extended primary

An extended primary is a shorthand syntactic notation. Apart from the special syntactic form, an extended primary has no special properties and denotes an operation and its parameter(s). The application of `Extract` operations (for indexing), field extract operations for structure and choice data type, and the `Make` operation are the canonical form, for example: `a[i]` becomes `Extract(a,i);` `s.f1` becomes `f1Extract(s);` and `(.x,y.)` becomes `Make(x,y).`

Concrete 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>

NOTE 1 – The square bracket form is preferred because it is distinct from parentheses used for expressions or operation applications. The two forms are otherwise equivalent.

<field primary> ::=
<primary> <exclamation mark> <field name>
| <primary> <full stop> <field name>

NOTE 2 – The <full stop> form is most similar to field selection in other languages, but is not always as obvious and distinct as use of an <exclamation mark>, so the latter is preferred. The two forms are otherwise equivalent.

<field name> ::=
<field name>

NOTE 3 – Each <field name> has to be a <name>, whereas in SDL-2000 it is a name or a number (an <integer name> or a <real name>).

<composite primary> ::=
[<qualifier>] <composite begin sign> <actual parameter list> <composite end sign>

The <actual parameter list> of an <extended primary> corresponds to the application of operations and therefore it is not allowed to omit any parameters.

Model

An <indexed primary> is derived concrete syntax for:
`Extract` **(** <primary> , <actual parameter list> **)**

The abstract syntax is determined from this concrete expression according to clause 12.2.1. For example, `a[i][j]` becomes `Extract(Extract(a,i),j)`.

NOTE 4 – `Extract` is defined for the `Predefined` data types `String`, `Charstring`, `Array`, `Vector`, `Bitstring` and `Octetstring`. For these types it has two parameters for the primary expression, and the index of the element to be extracted.

A <field primary> is derived concrete syntax for:

field-extract-name **(** <primary>)

where the *field-extract-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 clause 12.2.1.

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 clause 12.2.1.

12.2.4 Equality expression

Abstract grammar

Equality-expression	= *Positive-equality-expression* | *Negative-equality-expression*	
Positive-equality-expression	::	*First-operand* *Second-operand*
Negative-equality-expression	::	*First-operand* *Second-operand*
First-operand	=	*Expression*
Second-operand	=	*Expression*

An *Equality-expression* represents the equality of either values or identities of its *First-operand* and its *Second-operand*.

Concrete 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. An <equality expression> using the <equals sign> represents a *Positive-equality-expression*. An <equality expression> using the <not equals sign> represents a *Negative-equality-expression*. The <operand2> represents a *First-operand*, and the <operand3> represents a *Second-operand*.

Semantics

The *Equality-expression* returns a predefined `Boolean` `true` or `false`, and has an aggregation kind **PART**. The *Negative-equality-expression* returns `false` if and only if the *Positive-equality-expression* for the same operands returns `true`.

Interpretation of the *Equality-expression* proceeds by interpretation of its *First-operand* and its *Second-operand*.

If, after interpretation, both operands are pids (that is, a pid sort or the `Pid` sort), then the *Equality-expression* denotes identity between agents. The *Positive-equality-expression* returns the predefined `Boolean` value `true` if and only if both operands are either `Null` or identify the same agent instance.

If, after interpretation, both of the operands are values and the aggregation kind of one of the operands is **PART**, the *Equality-expression* denotes equality of values. The *Positive-equality-expression* returns the predefined `Boolean` value `true` if both operands are "undefined" (they access a variable which has no value associated), and predefined `Boolean` value `false` if only one operand is "undefined". Otherwise, the *Positive-equality-expression* returns the result of the application of the `equal` operator to *First-operand* and *Second-operand*, where `equal` corresponds to an *Operation-signature* with its *Operation-name* derived from `equal`, two *Formal-argument* items of the value sort that is compatible with the two operands, and a result being the predefined `Boolean` sort.

12.2.5 Conditional expression

A conditional expression is an expression where a Boolean expression is evaluated to determine whether to interpret a consequence or an alternative expression.

Abstract grammar

Conditional-expression	::	*Boolean-expression* *Consequence-expression* *Alternative-expression*
Boolean-expression	=	*Expression*
Consequence-expression	=	*Expression*
Alternative-expression	=	*Expression*

The sort of the *Consequence-expression* shall be the same as the sort of the *Alternative-expression*.

Concrete grammar

<conditional expression> ::=
    if <Boolean expression>
    then <consequence expression>
    else <alternative expression>
    fi

<consequence expression> ::=
    <expression>

<alternative expression> ::=
    <expression>

Semantics

The *Boolean-expression* is interpreted and either the *Consequence-expression* or the *Alternative-expression* is interpreted.

If the *Boolean-expression* returns the predefined `Boolean` value `true`, the *Alternative-expression* is not interpreted. If the *Boolean-expression* returns the predefined `Boolean` value `false`, the *Consequence-expression* is not interpreted.

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*).

12.2.6 Operation application

Abstract grammar

Operation-application	::	*Operation-identifier* *Actual-parameters*

Operation-identifier = *Identifier*

The *Operation-identifier* denotes an *Operation-signature*. Each *Expression* in the list of *Expression* of the *Actual-parameters* of the *Operation-application* shall be sort compatible with the corresponding (by position) sort in the *Formal-argument* list of the *Operation-signature*. There shall be no **UNDEFINED** elements in the *Actual-parameters* of the *Operation-application*.

Each *Operation-signature* has associated a *Procedure-definition*, as described in clause 12.1.7.

Each *Expression* of the *Actual-parameters* corresponding by position to an *Inout-parameter* or *Out-parameter* in the *Procedure-definition* associated with the *Operation-signature* shall be a *Variable-identifier* having the same *Sort-reference-identifier* as the corresponding (by position) sort in the *Formal-argument* list of the *Operation-signature*.

Concrete grammar

<operation application> ::=
    <operator application>

<operator application> ::=
    <operation identifier> [<actual parameters>]

If the <operation identifier> of the <operator application> is a <quoted operation name> that is <quotation mark> <equals sign> <quotation mark> or is <quotation mark> <not equals sign> <quotation mark>, and there are exactly two <actual parameters> where one <expression> of the <actual parameters> is sort compatible to the sort of the other <expression> of the <actual parameters>, the <operation application> represents an *Equality-expression* (see clause 12.2.4). The two <expression> items of the <actual parameters> represent *First-operand* and *Second-operand* of the *Equality-expression.* For an <equals sign> the *Equality-expression* is a *Positive-equality-expression*, and for <not equals sign> the *Equality-expression* is a *Negative-equality-expression*.

NOTE – *Equality-expression* uses the generic equal operator. It is not allowed to define an operation with two parameters and an <operation name> that is <quotation mark> <equals sign> <quotation mark> or <quotation mark> <not equals sign> <quotation mark>.

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 clause 6.2). 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. If there are two operations with the same <name> but differing by one or more of the argument or result sorts, each has a different *Operation-name*.

It shall 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.

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 clause 12.2.1.

It is allowed to omit <actual parameters> in an <operation application> only if the operation has no parameters.

Semantics

Resolution by context (see clause 6.2) guarantees that an operation is selected, such that the types of the actual arguments are pairwise sort compatible with the types of the formal arguments.

If an *Operation-application* has an *Operation-identifier* that identifies a *Static-operation-signature* (always the case in Basic SDL-2010), the *Operation-application* is interpreted as a *Value-returning-call-node* by invoking the *Procedure-definition* identified by the *Procedure-identifier* of the *Operation-signature*. This *Value-returning-call-node* has the *Procedure-identifier* of the *Procedure-definition* and the same *Actual-parameters* as the *Operation-application*. The abstract grammar rules and semantics of *Value-returning-call-node* in clause 11.13.3 apply. The procedure graph (derived from the operation diagram) is interpreted as explained in clause 9.4.

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.

An *Operation-application* has a sort, which is the sort of the result obtained by the interpretation of the procedure.

The aggregation kind of an *Operation-application* is the *Result-aggregation* of the procedure interpreted.

If the result sort of the operation signature is a syntype, then the range check defined in clause 12.1.8.2 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` is raised.

12.2.7 Range check expression

A range check expression checks if an expressions is within the range of values given by a sort and a constraint, or the range of values of a sort (usually a syntype if the range check expression is to be useful).

Abstract grammar

Range-check-expression :: *Expression Parent-sort-identifier Range-condition*

The sort of the *Expression* of a *Range-check-expression* shall be sort compatible with the sort identified by the *Parent-sort-identifier*, or the sort `Pid` if the *Parent-sort-identifier* is a pid sort.

Concrete grammar

<range check expression> ::=
<operand2> **in type** { <range check constrained sort> | <syntype> | <pid sort> }

<range check constrained sort> ::=
<sort identifier> <constraint>

The <sort identifier> of a <range check constrained sort> shall not be `Pid` or a pid sort.

The <operand2> represents the *Expression*. If the form <range check constrained sort> is used, the <sort identifier> represents the *Parent-sort-identifier* that applies to the <constraint>, which represents the *Range-condition* as described in clause 12.1.8.2. If the <sort identifier> in <range check constrained sort> identifies a syntype, this is the same as specifying the parent sort identifier of the syntype, and the <constraint> is not restricted to the range of the syntype. If the form <syntype> is used, *Parent-sort-identifier* and *Range-condition* are the *Parent-sort-identifier* and *Range-condition* (respectively) of the identified syntype. If the form <pid sort> is used, it determines the *Parent-sort-identifier* and the *Range-condition* is empty.

Semantics

A *Range-check-expression* is an expression of the predefined `Boolean` sort. If the *Range-condition* is empty and the *Parent-sort-identifier* identifies a pid sort, the *Range-check-expression* has the result `true` if and only if the *Expression* identifies an agent instance that is compatible with the *Parent-sort-identifier*; otherwise, it has the result `false`. Otherwise (*Range-condition* is not empty or the *Parent-sort-identifier* does not identify a pid sort), the *Range-check-expression* has the result `true` if the result of the *Expression* fulfils the *Range-condition* as described in clause 12.1.8.2; otherwise, it has the result `false`. A *Range-check-expression* has an *Aggregation-kind* of **PART**.

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. By assigning a new data item to the variable, the data item associated with a variable is changed. The data item associated with the variable is 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 varies according to the data item last assigned to the variable. The result of interpreting an active expression depends 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* *Aggregation-kind* [*Constant-expression*]
Variable-name	=	*Name*

If the *Constant-expression* is present, it shall be sort compatible with the *Sort-reference-identifier* denoted.

If *Sort-reference-identifier* is a *Syntype-identifier* and *Constant-expression* is present, the result of the *Constant-expression* shall be valid for the *Range-condition* of the syntype.

Aggregation-kind	=	**PART**

NOTE 1 – *Aggregation-kind* is always a **PART** in Basic SDL-2010 and for data that are not object-oriented. It is used to determine how assignment is interpreted. Data holders (variables, procedure parameters, procedure results) and expressions have an aggregation kind. *Aggregation-kind* is introduced so that alternatives to **PART** for object-oriented data, and therefore alternative interpretations of assignment, are possible.

Concrete grammar

<variable definition> ::=
dcl <variables of sort> {**,** <variables of sort> }* <end>

<variables of sort> ::=
<aggregation kind> <<u>variable</u> name> { **,** <<u>variable</u> name> }*
<sort> [<is assigned sign> <constant expression>]

NOTE – Each <<u>variable</u> name> has to be a <name>, whereas in SDL-2000 it is a name or a number (an <integer name> or a <real name>).

There is a *Variable-definition* for each <<u>variable</u> name> in <variables of sort>.

<aggregation kind> ::=
{ }

An empty <aggregation kind> represents an *Aggregation-kind* of **PART**.

NOTE − In Basic SDL-2010 <aggregation kind> is always empty.

The <aggregation kind> represents the *Aggregation-kind* of the *Variable-definition*. The <sort> represents the *Sort-reference-identifier* of the *Variable-definition*.

The *Constant-expression* is represented by:

a) if a <constant expression> is given in the <variable definition>, then this <constant expression> for each of the variables in the same <variables of sort>;

b) else, if the data type that defined the <sort> has a <default initialization>, then the <constant expression> of the <default initialization> as described in clause 12.3.3.2;

c) else, if the sort is a pid sort or the sort `Pid`, the *Constant-expression* is the *Null-literal-signature* for the sort.

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

Semantics

The *Aggregation-kind* of a *Variable-definition* determines what happens when an assignment to the variable is interpreted.

When a variable is created and the *Constant-expression* is present, then the variable is associated with the result of the *Constant-expression*.

Otherwise, if no *Constant-expression* applies, the variable has no data item associated: that is, the variable is "undefined".

12.3.2 Variable access

Abstract grammar

Variable-access :: *Variable-identifier*

Concrete grammar

<variable access> ::=
<<u>variable</u> identifier>

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.

Provided the variable is not "undefined", a variable access has a result, which is the data item last associated with the variable. If the variable is "undefined", the predefined exception `UndefinedVariable` is raised when the variable is accessed, except in the case the *Variable-access* is the *First-operand* or *Second-operand* of an *Equality-expression*.

A variable access has an aggregation kind, which is the *Aggregation-kind* of the *Variable-definition* identified by the *Variable-identifier*.

12.3.3 Assignment

An assignment creates an association from the variable to the result of interpreting an expression.

Abstract grammar

Assignment :: *Variable-identifier*
Expression

In an *Assignment*, the sort of the *Expression* shall be sort compatible with the sort of the *Variable-identifier*.

If the variable is declared with a *Sort-reference-identifier* that is a *Syntype-identifier* and the *Expression* is a *Constant-expression*, the result of the range check defined in clause 12.1.8.2 applied to the *Expression* shall be the predefined Boolean value true.

Concrete 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 defined in clause 12.3.3.1.

Semantics

An *Assignment* is interpreted as creating an association from the variable identified in the assignment with the result of the expression in the assignment. The previous association of the variable is lost.

The sort of the variable is the sort identified by the *Sort-reference-identifier* of the *Variable-definition* identified by the *Variable-identifier*. This is the *Sort* of a *Value-data-type-definition* or *Interface-definition* if the *Sort-reference-identifier* is a *Sort-identifier*, and is a syntype with a *Syntype-name* if the *Sort-reference-identifier* is a *Syntype-identifier*. For an *Interface-definition* the sort is a pid sort.

The *Aggregation-kind* of the variable is the *Aggregation-kind* of the *Variable-definition* identified by the *Variable-identifier*.

If the sort of the variable is a syntype, the range check defined in clause 12.1.8.2 is applied to the result of the *Expression*. If this range check returns the predefined `Boolean` value `false`, the predefined exception `OutOfRange` is raised and the variable has no data item associated: that is, the variable is "undefined".

If the sort of the variable is a pid sort and the result of the *Expression* identifies an agent instance that is not compatible with the pid sort of the variable, the predefined exception `OutOfRange` is raised and the variable has no data item associated: that is, the variable is "undefined".

Provided the predefined exception `OutOfRange` is not raised, the variable is associated with the result of the *Expression*. The manner in which this association is established depends on the sort and *Aggregation-kind* of the variable, and the sort (and *Aggregation-kind* if there is object oriented data) of the *Expression*:

a) If the sort of the variable is the *Sort* of a *Value-data-type-definition*, then
   1) if the *Aggregation-kind* of variable is **PART**, a copy of the value returned by the result of the *Expression* is associated with the identified variable.
b) If the sort of the variable is the `Pid` sort or a pid sort (the *Sort* of an *Interface-definition*) and the result of the *Expression* is a pid (it identifies an agent instance), the *Variable-identifier* is associated with the pid that is the result of *Expression*.

12.3.3.1 Extended variable

An extended variable allows the target of an assignment to be an indexed variable for data types that have indexed elements (such as strings, vectors or arrays) or field variables (for structure and choice data types). The effect of using an extended variable in an assignment is given by a description, where for an expression a complete composite variable (that is, the complete string, vector, array, structure or choice) is constructed and then assigned to the composite variable.

Concrete grammar

<extended variable> ::=
<indexed variable>
| <field variable>

<indexed variable> ::=
<variable> **(** <actual parameter list> **)**
| <variable> <left square bracket> <actual parameter list> <right square bracket>

NOTE 1 – The <actual parameter list> of an <indexed variable> corresponds to an operation application, therefore it is not allowed to omit any parameters.

NOTE 2 – The square bracket form is preferred because it is distinct from parentheses used for expressions or operation applications. The two forms are otherwise equivalent.

<field variable> ::=
<variable> <exclamation mark> <field name>
| <variable> <full stop> <field name>

NOTE 3 – The <full stop> form is most similar to field selection in other languages, but is not always as obvious and distinct as use of an <exclamation mark>, so the latter is preferred. The two forms are otherwise equivalent.

Model

The concrete syntax alternative of <indexed variable>:
<variable> **(** <actual parameter list> **)** <is assigned sign>

is equivalent to the alternative using square brackets.

The concrete syntax form:
<variable> <left square bracket> <actual parameter list> <right square bracket> <is assigned sign> <expression>

is derived concrete syntax for:
<variable> <is assigned sign> `Modify` **(** <variable> , <actual parameter list>, <expression> **)**

where the parameter list for `Modify` is constructed by adding <variable> before the <actual parameter list>, and <expression> after the <actual parameter list>.

The abstract grammar is determined from this concrete expression according to clause 12.2.1. The same model applies to the first form of <indexed variable>.

NOTE 4 – `Modify` is defined for the `Predefined` data types `String`, `Charstring`, `Array`, `Vector`, `Bitstring` and `Octetstring`. For these types it has three parameters for the variable to be modified, as well as the index and the new value for the modified element.

The concrete syntax form:
<variable> <exclamation mark> <field name> <is assigned sign> <expression>

is derived concrete syntax for:
<variable> <is assigned sign> *field-modify-name* **(** <variable>, <expression> **)**

where the *field-modify-name* is formed from the concatenation of the field name and "`Modify`". The abstract syntax is determined from this concrete expression according to clause 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.

Abstract grammar

Default-initialization = *Constant-expression*

Concrete grammar

<default initialization> ::=
default <constant expression>

Semantics

A *Default-initialization* is applied as the *Constant-expression* of a *Variable-definition* only if the *Variable-definition* does not have an explicit constant expression.

The *Default-initialization* of a *Data-type-definition* represents the *Constant-expression* of any otherwise un-initialized *Variable-definition* of the sort defined by the *Data-type-definition*.

The *Default-initialization* of a *Syntype-definition* represents the *Constant-expression* of any otherwise un-initialized *Variable-definition* of the sort defined by the *Syntype-definition*.

If no *Default-initialization* is given in the *Syntype-definition* for a sort but the *Data-type-definition* for the parent sort has a *Default-initialization* and this is in the range defined for the syntype, this represents the *Constant-expression* of any otherwise un-initialized *Variable-definition* of the sort. If the *Default-initialization* is not in the range defined for the syntype, the *Constant-expression* is omitted (see clause 12.3.1).

12.3.4 Imperative expression

Imperative expressions obtain results from the underlying system state.

Abstract grammar

```
Imperative-expression       =    Now-expression
                            |    Pid-expression
                            |    Timer-active-expression
                            |    Timer-remaining-duration
                            |    Active-agents-expression
```

Concrete grammar

```
<imperative expression> ::=
                 <now expression>
            |    <pid expression>
            |    <timer active expression>
            |    <timer remaining duration>
            |    <active agents expression>
```

Imperative expressions are expressions for accessing the system clock, the result of imported variables, the pid associated with an agent and the status of timers.

12.3.4.1 Now expression

Abstract grammar

```
Now-expression              ::   { }
```

Concrete 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. Unless otherwise specified, the time unit is 1 second. 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 Pid expression

Abstract grammar

Pid-expression	=	*Self-expression*
	|	*Parent-expression*
	|	*Offspring-expression*
	|	*Sender-expression*
Self-expression	::	{ }
Parent-expression	::	{ }
Offspring-expression	::	{ }
Sender-expression	::	{ }

Concrete grammar

<pid expression> ::=
<self expression>
| <parent expression>
| <offspring expression
| <sender expression>

<self expression> ::=
self

<parent expression> ::=
parent

<offspring expression> ::=
offspring

<sender expression> ::=
sender

Semantics

A *Pid-expression* accesses one of the implicit anonymous variables self, parent, offspring or sender (see clause 9). A *Pid-expression* has a result, which is the last pid associated with the corresponding implicit variable.

A *Parent-expression*, *Offspring-expression*, *Sender-expression* or *Self-expression* has a static sort, as defined in clause 9.

12.3.4.3 Timer active expression and timer remaining duration

Abstract grammar

Timer-active-expression	::	*Timer-identifier* *Expression**
Timer-remaining-duration	::	*Timer-identifier* *Expression**

The sorts of the *Expression* list in the *Timer-active-expression* or *Timer-remaining-duration* shall correspond by position to the *Sort-reference-identifier* list directly following the *Timer-name* (see clause 11.15) identified by the *Timer-identifier*.

Concrete grammar

<timer active expression> ::=
active (<timer identifier> [**(** <expression list> **)**] **)**

<timer remaining duration> ::=
rem (<timer identifier> [**(** <expression list> **)**] **)**

Semantics

The timer of a *Timer-active-expression* or *Timer-remaining-duration* is the timer identified by *Timer-identifier* and set with the same results as denoted by the *Expression* list (if any). The expressions are interpreted in the order given.

If a sort specified in a timer definition is a syntype, the range check defined in clause 12.1.8.2 applied to the corresponding *Expression* in the *Expression* list of the *Timer-active-expression* or *Timer-remaining-duration* shall be the predefined `Boolean` value `true`; otherwise, the predefined exception `OutOfRange` is raised.

A *Timer-active-expression* is an expression of the predefined `Boolean` sort, which has the result `true`, if the timer is active (see clause 11.15). Otherwise, the *Timer-active-expression* has the result `false`.

A *Timer-remaining-duration* is an expression of the predefined `Duration` sort. The result value for an active time is the duration before the timer is due to expire, which is the time the timer was last set to minus **now**. The duration will be negative if the timer is active but has already expired. If the timer is inactive, the value is zero (which can be distinguished from an active time returning zero by a subsequent timer active expression).

12.3.4.4 Active agents expression

An active agents expression gives the current number of agent instances in an agent instance set.

NOTE – This enables comparison of the current number of instances with the *Lower-bound* for the instance set, so that it is possible to avoid interpretation of a stop if the number of instances is equal to the *Lower-bound*. However, if the agent instance set is not contained within a process, it is still possible that between interpreting the active agents expression and interpreting a stop, another instance in the set interprets a stop with the number of instances already equal to the *Lower-bound*.

Abstract grammar

Active-agents-expression :: { *Agent-identifier* | **THIS** }

Concrete grammar

<active agents expression> ::=
active ({ <agent identifier> | **this** } **)**

this shall only be specified in an <agent type diagram> and in scopes enclosed by an <agent type diagram> and represents **THIS**.

Semantics

An *Active-agents-expression* is an expression of the predefined `Natural` sort. The result value is the current number of instances in the agent instance set identified by the *Agent-identifier* or **THIS**.

THIS identifies the set of instances of the agent in which the *Active-agents-expression* is being interpreted.

12.3.5 Value returning procedure call

The abstract grammar for a value returning procedure call and static semantic constraints are shown in clause 11.13.3.

Concrete grammar

<value returning procedure call> ::=
[**call**] <procedure call body>

It is not allowed to omit keyword **call** if the <value returning procedure call> is syntactically ambiguous with an operation (or variable) with the same name followed by a parameter list.

NOTE – This ambiguity is not resolved by context.

The <procedure identifier> in a <value returning procedure call> shall identify a procedure having a <procedure result>.

The <procedure call body> represents a *Value-returning-call-node*, where *Procedure-identifier* is represented by the <procedure identifier>, and the *Expression* list is represented by the list of actual parameters. The semantics of the *Value-returning-call-node* is shown in clause 11.13.3.

SERIES OF ITU-T RECOMMENDATIONS

Series A	Organization of the work of ITU-T
Series D	Tariff and accounting principles and international telecommunication/ICT economic and policy issues
Series E	Overall network operation, telephone service, service operation and human factors
Series F	Non-telephone telecommunication services
Series G	Transmission systems and media, digital systems and networks
Series H	Audiovisual and multimedia systems
Series I	Integrated services digital network
Series J	Cable networks and transmission of television, sound programme and other multimedia signals
Series K	Protection against interference
Series L	Environment and ICTs, climate change, e-waste, energy efficiency; construction, installation and protection of cables and other elements of outside plant
Series M	Telecommunication management, including TMN and network maintenance
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Telephone transmission quality, telephone installations, local line networks
Series Q	Switching and signalling, and associated measurements and tests
Series R	Telegraph transmission
Series S	Telegraph services terminal equipment
Series T	Terminals for telematic services
Series U	Telegraph switching
Series V	Data communication over the telephone network
Series X	Data networks, open system communications and security
Series Y	Global information infrastructure, Internet protocol aspects, next-generation networks, Internet of Things and smart cities
Series Z	**Languages and general software aspects for telecommunication systems**

Printed in Switzerland
Geneva, 2019