

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Z.101

(04/2016)

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

Scope/Objective

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

Keywords

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

* 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/11.1002/1000/11388-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/>.

© ITU 2016

All rights reserved. No part of this publication may be reproduced, by any means whatsoever, without the prior written permission of ITU.

Table of Contents

		Page
1	Scope and objective	1
	1.1 Objective.....	1
	1.2 Application	1
2	References.....	2
3	Definitions	2
	3.1 Terms defined elsewhere	2
	3.2 Terms defined in this Recommendation.....	2
4	Abbreviations and acronyms	2
5	Conventions	2
6	General rules	3
	6.1 Lexical rules	3
	6.2 End terminator and comment	10
	6.3 Empty clause	10
	6.4 Solid association symbol	10
	6.5 The metasymbol <i>is followed by</i> and flow line symbols.....	10
	6.6 Names and identifiers, name resolution and visibility	11
	6.7 Empty clause	16
	6.8 Informal text	16
	6.9 Text symbol	17
	6.10 Frame symbol and page numbers	17
7	Organization of Specification and Description Language specifications.....	18
	7.1 Framework.....	18
	7.2 Package.....	19
	7.3 Referenced definition	21
8	Structural concepts.....	22
	8.1 Types, instances and gates.....	22
	8.2 Type references and operation references	28
9	Agents	29
	9.1 System	34
	9.2 Block.....	34
	9.3 Process.....	35
	9.4 Procedure	36
10	Communication.....	39
	10.1 Channel.....	39
	10.2 Connection.....	42
	10.3 Signal.....	42
	10.4 Signal list area	43

		Page
11	Behaviour.....	44
11.1	Start.....	44
11.2	State	44
11.3	Input.....	46
11.4	Empty clause	48
11.5	Empty clause	48
11.6	Empty clause	48
11.7	Save	48
11.8	Empty clause	49
11.9	Empty clause	49
11.10	Label (connector name).....	49
11.11	State machine and composite state	50
11.12	Transition.....	52
11.13	Action	57
11.14	Statement lists.....	64
11.15	Timer	65
12	Data.....	67
12.1	Data definitions	67
12.2	Use of data	88
12.3	Active use of data	97

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.

Copyright International Telecommunication Union/ITU Telecommunication Sector
Provided by IHS Markit under license with ITU-T
No reproduction or networking permitted without license from IHS

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 (2016), *Specification and Description Language – Overview of SDL-2010*.
- [ITU-T Z.102] Recommendation ITU-T Z.102 (2016), *Specification and Description Language – Comprehensive SDL-2010*.
- [ITU-T Z.103] Recommendation ITU-T Z.103 (2016), *Specification and Description Language – Shorthand notation and annotation in SDL-2010*.
- [ITU-T Z.104] Recommendation ITU-T Z.104 (2016), *Specification and Description Language – Data and action language in SDL-2010*.
- [ITU-T Z.105] Recommendation ITU-T Z.105 (2016), *Specification and Description Language – SDL-2010 combined with ASN.1 modules*.
- [ITU-T Z.106] Recommendation ITU-T Z.106 (2016), *Specification and Description Language – Common interchange format for SDL-2010*.
- [ITU-T Z.107] Recommendation ITU-T Z.107 (2016), *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> ::=	<table><tr><td><exclamation mark></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr><tr><td><left parenthesis></td><td></td><td></td><td><right parenthesis></td><td></td><td></td><td></td><td></td></tr><tr><td><plus sign></td><td></td><td></td><td><comma></td><td></td><td></td><td><hyphen></td><td></td></tr><tr><td><full stop></td><td></td><td></td><td><colon></td><td></td><td></td><td></td><td></td></tr><tr><td><less than sign></td><td></td><td></td><td><equals sign></td><td></td><td></td><td><greater than sign></td><td></td></tr><tr><td><left square bracket></td><td></td><td></td><td><right square bracket></td><td></td><td></td><td></td><td></td></tr><tr><td><left curly bracket></td><td></td><td></td><td><right curly bracket></td><td></td><td></td><td></td><td></td></tr></table>							<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>				
<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> ::=	<table><tr><td><quotation mark></td><td></td><td><dollar sign></td><td></td><td><percent sign></td><td></td><td></td><td></td></tr><tr><td><ampersand></td><td></td><td><question mark></td><td></td><td><commercial at></td><td></td><td></td><td></td></tr><tr><td><reverse solidus></td><td></td><td><circumflex accent></td><td></td><td><underline></td><td></td><td></td><td></td></tr><tr><td><grave accent></td><td></td><td><vertical line></td><td></td><td><tilde></td><td></td><td></td><td></td></tr></table>							<quotation mark>		<dollar sign>		<percent sign>				<ampersand>		<question mark>		<commercial at>				<reverse solidus>		<circumflex accent>		<underline>				<grave accent>		<vertical line>		<tilde>																											
<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>	::=	}																																																													

Copyright International Telecommunication Union/ITU-Telcommunication Sector

<number sign>	::=	#
<dollar sign>	::=	\$
<percent sign>	::=	%
<ampersand>	::=	&
<apostrophe>	::=	'
<question mark>	::=	?
<commercial at>	::=	@
<reverse solidus>	::=	\
<circumflex accent>	::=	^
<underline>	::=	_
<grave accent>	::=	`
<vertical line>	::=	
<tilde>	::=	~

Copyright International Telecommunication Union

<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 "`dc1 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.

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>` 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.

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.

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.

Abstract grammar

		<i>Procedure-qualifier</i>
		<i>Interface-qualifier</i>
<i>Package-qualifier</i>	::	<i>Package-name</i>
<i>Agent-type-qualifier</i>	::	<i>Agent-type-name</i>
<i>Agent-qualifier</i>	::	<i>Agent-name</i>
<i>State-type-qualifier</i>	::	<i>State-type-name</i>
<i>State-qualifier</i>	::	<i>State-name</i>
<i>Data-type-qualifier</i>	::	<i>Data-type-name</i>
<i>Procedure-qualifier</i>	::	<i>Procedure-name</i>
<i>Interface-qualifier</i>	::	<i>Interface-name</i>
<i>Package-name</i>	=	<i>Name</i>
<i>Agent-type-name</i>	=	<i>Name</i>
<i>Agent-name</i>	=	<i>Name</i>
<i>State-type-name</i>	=	<i>Name</i>
<i>Data-type-name</i>	=	<i>Name</i>
<i>Interface-name</i>	=	<i>Name</i>
<i>Name</i>	::	<i>Token</i>

The *Path-item* list of a *Qualifier* shall give the full path to the identified entity.

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>. Therefore, it is not possible to refer to the identifiers introduced in a definition attached to these scope units by qualifiers.

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 diagram> or <process type diagram>.

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.

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>.

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.

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*.

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>, 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 except any outermost <package diagram> 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>s 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> shall be present if the <referenced definition> is a <package diagram> referenced from another context, otherwise 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*.

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

<i>Agent-type-definition</i>	::	<i>Agent-type-name</i> <i>Agent-kind</i> [<i>Agent-type-identifier</i>] <i>Agent-formal-parameter</i> * <i>Data-type-definition-set</i> <i>Syntype-definition-set</i> <i>Signal-definition-set</i> <i>Timer-definition-set</i> <i>Variable-definition-set</i> <i>Agent-type-definition-set</i> <i>Composite-state-type-definition-set</i> <i>Procedure-definition-set</i> <i>Agent-definition-set</i> <i>Gate-definition-set</i> <i>Channel-definition-set</i> <i>State-machine</i>
------------------------------	----	--

<i>Agent-kind</i>	=	SYSTEM BLOCK PROCESS
<i>Agent-type-identifier</i>	=	<i>Identifier</i>
<i>Agent-formal-parameter</i>	=	<i>Parameter</i>
<i>Parameter</i>	::	<i>Variable-name</i> <i>Sort-reference-identifier</i> <i>Parameter-aggregation</i>
<i>Parameter-aggregation</i>	::	<i>Aggregation-kind</i>
<i>State-machine</i>	::	<i>State-name</i> <i>Nextstate-parameters</i> <i>Composite-state-type-identifier</i>

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> }
    [ is associated with <package use area> ]
```

The <package use area> shall be placed immediately above the <frame symbol> of the <agent 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> ::=
 <variable name> {, <variable name>}* <sort>

NOTE 5 – The <variable 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 <variable 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 <variable 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.

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> }*

<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**.

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> }*

<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**.

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> }*

<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**.

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*

NOTE 1 – The *Gate-definition-set* of a *Composite-state-type-definition* should normally be empty, except when the *Composite-state-type-definition* is identified by the *State-machine* of an agent, in which case each *Gate-definition* defines a gate of the state machine. In this case, channels are used to connect the gates of the state machine to the external gates of the agent.

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 2 – 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 3 – 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>

<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

<i>Gate-definition</i>	::	<i>Gate-name</i>
		<i>In-signal-identifier-set</i>
		<i>Out-signal-identifier-set</i>

Gate-name = *Name*

$$In\text{-}signal\text{-}identifier = Signal\text{-}identifier$$
$$Out\text{-}signal\text{-}identifier = Signal\text{-}identifier$$

Concrete grammar

$$\langle \text{gate on diagram} \rangle ::= \langle \text{gate definition} \rangle$$

```

<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

```

$$\langle \text{gate symbol } 1 \rangle ::=$$

$$\langle \text{gate symbol 2} \rangle ::=$$

$$\langle \text{gate} \rangle ::= \langle \text{gate name} \rangle$$

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*