International Telecommunication Union

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Z.102
(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 – Comprehensive SDL-2010

Recommendation ITU-T Z.102

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.102

Specification and Description Language – Comprehensive SDL-2010

Summary

Recommendation ITU-T Z.102 defines the comprehensive features of the Specification and Description Language. Together with Recommendations ITU-T Z.100, ITU-T Z.101, 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 features of the language not included in Basic SDL-2010 in Recommendation ITU-T Z.101. These features provide comprehensive coverage of abstract grammar of the language except some data features covered in ITU-T Z.104 (and ITU-T Z.107 for object-oriented data).

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 additional features to Recommendation ITU-T Z.101. 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 with Recommendations ITU-T Z.101 and ITU-T Z.104 (and ITU-T Z.107 for object-oriented data) gives a canonical syntax, which is extended in ITU-T Z.103 to a syntax that is easier to use. 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 applications is from requirement description to implementation. The features of the language defined in Recommendation ITU-T Z.102 allow more complex models to be defined than is practical with ITU-T Z.101, but without the features of Recommendation ITU-T Z.103 that make the language more concise and informative. This Recommendation provides a basis for the features defined in Recommendation ITU-T Z.103, which (with ITU-T Z.104 for data and ITU-T Z.107 for object-oriented data) completes the main language feature set.

History

Edition	Recommendation	Approval	Study Group	Unique ID[*]
1.0	ITU-T Z.102	2011-12-22	17	11.1002/1000/11389
2.0	ITU-T Z.102	2016-04-29	17	11.1002/1000/12856
3.0	ITU-T Z.102	2019-10-14	17	11.1002/1000/14053

Keywords

Canonical syntax, comprehensive SDL-2010, context parameters, generic systems, inheritance, remote procedures, remote variables, SDL-2010, Specification and Description Language, specialization, state aggregation.

[*] 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.

Table of Contents

Page

Introduction

Status/Stability

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.102

Specification and Description Language – Comprehensive SDL-2010

1 Scope and objective

This Recommendation defines features of the Specification and Description Language that extend the features of the Basic Specification and Description Language defined in [ITU-T Z.101] to comprehensively cover the abstract grammar of the language. A canonical concrete syntax is given to cover the abstract grammar. The abstract grammar defined in this document with the abstract grammar of the Basic Specification and Description Language comprehensively defines properties of the language except the details of data and expressions. This Recommendation also includes features for inheritance of types, generic systems, macros and the handling of the Universal Multiple-Octet Coded Character Set. Together with [ITU-T Z.100], [ITU-T Z.101], [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 features of the Specification and Description Language additional to those features defined for Basic SDL-2010 so that the abstract grammar of all SDL-2010 except data and expressions is defined. As well as completing the abstract grammar, the features for the inheritance of types is added. The defined concrete syntax is a canonical form that closely matches the abstract grammar. While this makes it possible to use each of the features, a separate Recommendation on shorthand notation provides additional concrete grammar that makes SDL-2010 more practical to use.

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* sections, so that a model written using only Comprehensive SDL-2010 or Basic SDL-2010 is not as concise or as readable as one using the full language. Shorthand notation or *Model* sections in Comprehensive SDL-2010 are transformed into the canonical concrete syntax given for Comprehensive SDL-2010 or Basic SDL-2010 or the data and action language in SDL-2010.

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.101]	Recommendation ITU-T Z.101 (2019), *Specification and Description Language – Basic 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*.
[ISO/IEC 10646]	ISO/IEC 10646:2017, *Information technology – Universal Coded Character Set (UCS).*

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 Term 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, which includes the conventions defined in [ITU-T Z.111].

Where an abstract or concrete syntax rule is defined in this Recommendation with the same name as a rule in [ITU-T Z.101] the rule given here replaces the rule in [ITU-T Z.101]. Any *Abstract grammar* or *Concrete grammar* conditions, *Semantics* and *Model* defined on a named rule in [ITU-T Z.101] apply to the redefined rule, unless specifically defined otherwise in this Recommendation. Any contradiction between [ITU-T Z.101] and this Recommendation is an error in the definition of SDL-2010 that needs to be resolved by further study.

6 General rules

6.1 Lexical rules – UCS extended character set

The use of the extended character set of UCS (see [ISO/IEC 10646]) is allowed. UCS includes the International Reference Version (IRV) characters (see [ITU-T T.50] and [ITU-T Z.101]) as a subset.

If the UCS 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 the UCS extended character set that corresponds to an IRV <letter> is equivalent to the IRV <letter>. Similarly, a printing character of the UCS 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 the UCS extended character set that represents a letter in some script and does not correspond to an IRV <letter> is usable as a <letter>. Characters are treated in the order they occur in the source text, which possibly does not correspond to the apparent presentation or printing order depending on how UCS characters are handled for presentation or printing (such as right to left, left to right, or in combination). Non-printing UCS characters that do not correspond to an IRV character are treated in the same way as a control character is treated in [ITU-T Z.101].

6.2 End terminator and comment

Concrete grammar

An <end> in <statements> shall not contain <comment>.

6.3 Empty clause

This clause is intentionally left blank.

6.4 Solid association symbol

See [ITU-T Z.101].

6.5 The metasymbol is followed by and flow line symbols

See [ITU-T Z.101].

6.6 Visibility rules, names and identifiers – additional scope units

This Recommendation introduces concrete syntax for <compound statement>, which is a scope unit for the connector name and variables introduced in the statement.

A <signal definition> is a scope unit only for any formal context parameters of the defined signal. These context parameters are formal sorts that are replaced by actual sorts when the signal is specialized. Such specialization is not part of Basic SDL-2010 but is described in this Recommendation.

NOTE – A <sort context parameter> is not considered to be a scope unit in SDL-2010. It was listed as defining a scope unit in SDL-2000.

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

Abstract grammar

```
Path-item                    =    Package-qualifier
                             |    Agent-type-qualifier
                             |    Agent-qualifier
                             |    State-type-qualifier
```

```
                          |   State-qualifier
                          |   Data-type-qualifier
                          |   Procedure-qualifier
                          |   Interface-qualifier
                          |   Compound-node-qualifier

Compound-node-qualifier   ::  Interface-name

Compound-node-name        =   Name
```

The abstract syntax is extended from Basic SDL-2010 to include *Compound-node-qualifier* in *Path-item* to identify the scope of a compound statement.

Concrete grammar

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

The syntax of <scope unit kind> is extended from Basic SDL-2010 to include **composition** in Path-item to identify the scope of a compound statement. If the <scope unit kind> of a <qualifier> is **composition**, the <qualifier> a represents a *Compound-node-qualifier* and <name> of the <qualifier> is the <connector name> of a <compound statement>. If no <connector name> is given for the <compound statement>, a newly created anonymous name represents the *Connector-name*; therefore no explicit <qualifier> can be given. In this case the specification is only valid if all uses of a variable name defined in the compound statement are uniquely bound without a qualifier.

6.7 Macro

A macro definition contains a collection of lexical units, which are included as macro expansions in the places in the textual parts of the concrete grammar of an <sdl specification>. Each such place is indicated by a macro call. Before an <sdl specification> can be analysed, each macro call shall be replaced by the corresponding macro expansion.

6.7.1 Additional lexical rules

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

6.7.2 Macro definition

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

<macro formal parameters> ::=
    (<macro formal parameter> { **,** <macro formal parameter>}* **)**
  | **fpar** <macro formal parameter> {, <macro formal parameter>}*

NOTE – The two alternatives of <macro formal parameters> are equivalent.

<macro formal parameter> ::=
    <name>

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

<macro parameter> ::=
    <macro formal parameter>
  | **macroid**

Each <macro formal parameter> of a <macro definition> shall have a distinct <name> within the macro. Each <macro actual parameter> of a macro call shall be matched one to one by position with the corresponding <macro formal parameter>.

The <macro body> shall not contain the keyword **endmacro** or **macrodefinition**.

A <macro definition> contains lexical units.

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

A macro definition is allowed to contain macro calls, but a macro definition shall not call itself either directly or indirectly through macro calls in other macro definitions.

The keyword **macroid** is allowed as a pseudo macro formal parameter within each macro definition. No <macro actual parameter> item is allowed for the keyword **macroid** as a pseudo macro formal parameter, and the keyword is replaced by a unique <name> for each expansion of a macro definition (within an expansion, the same <name> is used for each occurrence of **macroid**).

Places that a <macro definition> is allowed are shown in *Concrete grammar*. A <macro definition> is not removed once macro expansion has taken place and therefore acts as a separator between other lexical units.

Example

Below is an example of a <macro definition>:

```
macrodefinition Exam (alfa, c, s, a);
     block type alfa referenced;
     dcl exported c as s Integer := a;
endmacro Exam;
```

6.7.3 Macro call

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

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

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

The <lexical unit> for a <macro actual parameter> is not allowed to be a comma "," or right parenthesis ")". If either of these two characters is required in a <macro actual parameter>, the <macro actual parameter> has to be a <character string>. If the <macro actual parameter> is a <character string>, the result of the <character string> is used when the <macro actual parameter> replaces a <macro formal parameter>.

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

Because a <macro call> is replaced by other lexical units once macro expansion has taken place, it does not occur in *Concrete grammar*.

Model

An <sdl specification> is allowed to contain macro definitions and macro calls. Before such an <sdl specification> is analysed for the remaining non-macro grammar, all macro calls shall be expanded. The expansion of a macro call means that a copy of the macro definition having the same <macro name> as that given in the macro call is expanded to replace the macro call. This means that a copy of the macro body is created, and each occurrence of the <macro formal parameter>s of the copy is replaced by the corresponding <macro actual parameter>s of the macro call, then macro calls in the copy, if any, are expanded. All percent characters (%) in <formal name>s are removed when <macro formal parameter>s are replaced by <macro actual parameter>s.

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

Example

Below is an example of a <macro call>, as a fragment inside a <text symbol> of a <block type diagram>.

```
.........
block type A referenced;
macro Exam (B, C1, S1, 12);
.........
```

The expansion of this macro call, using the example in clause 6.7.2, gives the following result.

```
.........
block type A referenced;
block type B referenced;
dcl exported C1 as S1 Integer := 12;
.........
```

7 Organization of Specification and Description Language specifications

7.1 Framework

See [ITU-T Z.101].

7.2 Package

Definitions as parts of a package are extended to include remote items and generic selection (in addition to defining types, signals, and interfaces as defined in Basic SDL-2010).

Concrete grammar

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

<package text area> is extended to cover <remote variable definition>, <remote procedure definition>, <select definition> and <macro definition>. Each <remote procedure definition> or <remote variable definition> in a <package text area> represents a member of the *Signal-definition-**set*** of the *Package-definition* for the implied signal of the remote procedure or variable. The content of a <macro definition> is used for macro calls, and the macro expansion represents the abstract grammar items in the context the macro call occurs. Only the selected items of a <select definition> represent items on the abstract grammar.

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

<diagram in package> is extended from Basic SDL-2010 to cover <option area> for generic specifications. Only the selected items of an <option area> represent items on the abstract grammar.

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

The <selected entity kind> used in <definition selection> of a <package use clause>, is extended from Basic SDL-2010 to include **remote procedure** and **remote** for remote procedures and remote variable definitions.

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

The keyword **remote** (not preceding **procedure**) is used for selection of a remote variable definition.

8 Structural concepts

Basic SDL-2010 is extended to include the generalization and specialization of types.

The language mechanisms introduced provide:

a) (pure) type definitions as defined in Basic SDL-2010;

b) typebased instance definitions that define instances or instance sets according to types as in Basic SDL-2010;

c) parameterized type definitions that are made (partially or completely) independent of the enclosing scope by means of context parameters and are bound to specific scopes by actual parameters;

d) specialization of supertype definitions into subtype definitions, by adding properties and by redefining virtual types and transitions.

8.1 Types, instances and gates

The type of an instance (or instance set) based on a parameterized type is the anonymous type formed by binding the parameters of the parameterized type to the actual parameters given.

8.1.1 Structural type definitions

Basic SDL-2010 is extended by the introduction of virtuality, context parameters and the state aggregation type.

8.1.1.1 Agent types

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
		[*Abstract*]

Agent-type-definition is extended to optionally allow *Abstract* to be specified.

Concrete grammar

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

<type preamble> is extended from the empty syntax in Basic SDL-2010 to optionally allow <virtuality> and <abstract> to be specified.

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

<agent type additional heading> is extended from Basic SDL-2010 to allow <formal context parameters> and <virtuality constraint> to be specified. If <agent type additional heading> has <formal context parameters>, the agent type is an abstract type: that is, *Abstract* is included in the *Agent-type-definition* (regardless of the keyword **abstract** in the <type preamble>).

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

<agent additional heading> is extended compared with Basic SDL-2010 to allow <specialization> for formal context parameters.

Semantics

The complete output set of an agent type includes in the union of all signals, the implicit signals for remote procedures and remote variables mentioned, either directly or as part of interfaces, in the outgoing signal lists associated with the gates of the agent type.

8.1.1.2 System type

Concrete grammar

A <formal context parameter> of <formal context parameters> of <agent type additional heading> of a <system type diagram> shall not be an <agent context parameter>, a <variable context parameter list> or a <timer context parameter list>.

8.1.1.3 Block type

See [ITU-T Z.101].

8.1.1.4 Process type

See [ITU-T Z.101].

8.1.1.5 Composite state type

Abstract grammar

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

Composite-state-type-definition is extended from Basic SDL-2010 to include an optional *Composite-state-type-identifier*, a *State-entry-point-definition-**set***, a *State-exit-point-definition-**set***, (as an alternative to *Composite-state-graph*), a *State-aggregation-node* and optionally *Abstract*. The optional *Composite-state-type-identifier* of *Composite-state-type-definition* identifies the base type (super type) of a specialization. See clause 8.1.3 for more information on *Abstract*.

Concrete grammar

```
<composite state type diagram> ::=
                <composite state type page>
            |   <state aggregation type page>
```

Basic SDL-2010 <composite state type diagram> is extended to allow <state aggregation type page> as an alternative to <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 connected to <state connection point area> }*
                [ is associated with <package use area> ]
```

Basic SDL-2010 <composite state type page> is extended to allow <state connection point area> connections.

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

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

<composite state type heading> is extended from Basic SDL-2010 to have optional

<formal context parameters>, optional <formal context parameters> and optional <specialization>.

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

8.1.2 Type expression

Type expression is extended from Basic SDL-2010, when the base type is a parameterized type to allow the type expression to denote new type formed by binding actual parameters to the base type. The use of type expression is extended from Basic SDL-2010 and it is used for defining one type in terms of another by specialization (see clause 8.4).

Concrete grammar

```
<type expression> ::=
                        <base type> [<actual context parameter list>]
```

<type expression> is extended from Basic SDL-2010 to allow <actual context parameter list>. It is valid to have <actual context parameter list> if and only if <base type> denotes a parameterized type. Context parameters are defined in clause 8.3. Outside a parameterized type, the parameterized type shall only be used by referring to its <identifier> in <type expression>.

Model

A <type expression> yields either the type identified by the identifier of <base type> in cases where there are no actual context parameters, or an anonymous type defined by applying the actual context parameters to the formal context parameters of the parameterized type denoted by the identifier of <base type>. The anonymous type definition is formed by:

1) copying the definition of the <base type> in the context where the construct using the <type expression> occurs and changing the name to an anonymous unique name;
2) replacing each occurrence of each <formal context parameter> name by the corresponding <actual context parameter> (if there is one) in the copy;
3) removing the <formal context parameter> from the <formal context parameter list> if there is a corresponding <actual context parameter>, and removing the <formal context parameters> if the <formal context parameter list> is empty;
4) replacing the <type expression> by a <type expression> with a <base type> that identifies the type with the anonymous unique name and no <actual context parameter list>.

NOTE 1 – Two textually identical <type expression>s with the <actual context parameter list> denote different types, because they have different anonymous names. To use the same type binding in different places, a type definition using the <type expression> should be used to name the <type expression>.

If some actual context parameters are omitted, the type is still parameterized.

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

NOTE 2 – The static properties on the usage of a <type expression> are possibly violated in the following cases, for example.

– When a scope unit has signal context parameters or timer context parameters, the condition that stimuli for a state have to be disjoint depends on the actual context parameters that will be used.
– When an output in a scope unit refers to a gate or a channel, which is not defined in the nearest enclosing type having gates, instantiation of that type results in an erroneous specification if there is no communication path to the gate.

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

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

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

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

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

– If a formal procedure context parameter is defined with an at least constraint and the actual context parameter has added a parameter of <parameter kind> in/out or out, it is possible that a call of the formal procedure context parameter in the parameterized type results in an omitted actual in/out or out parameter (that is, in an erroneous specification).

If the scope unit contains <specialization> and any <actual context parameter>s are omitted in the <type expression>, the <formal context parameter>s are copied (while preserving their order) and inserted in front of the <formal context parameter>s (if any) of the scope unit. In place of omitted <actual context parameter>s, the names of corresponding <formal context parameter>s are inserted as <actual context parameter>s. These <actual context parameter>s now have the defining context in the current scope unit.

8.1.3 Abstract type

A type is an abstract type if its definition contains *Abstract*. An abstract type is either defined as abstract by the keyword **abstract** or because it has unbound context parameters.

Abstract grammar

```
Abstract                    ::    {}
```

Abstract is part of the type definition. See clauses 8.1.1.1, 8.1.1.5, 9.4, 10.3 and 12.1.

Concrete grammar

```
<abstract> ::=
                    abstract
```

A type with unbound <formal context parameters> is also an abstract type.

NOTE – An abstract signal is allowed as a <signal constraint> in a <signal context parameter list>.

Semantics

An abstract type shall not be instantiated, therefore the *Abstract* property of a type prohibits instances of the type being used in the system, either from an explicit definition or because the type has unbound context parameters. However, instantiation of a subtype of an abstract data type is permitted, if the subtype is not itself abstract.

The *Abstract* property is not inherited: that is, whether a subtype is abstract depends on whether an explicit definition is given for the subtype, or if the subtype has unbound context parameters.

8.1.4 Gate

Gates are extended from Basic SDL-2010 to allow constraints to be specified on the types of agents that are connected (via a channel) to the gate, and allow the use of inherited gates in subtypes.

Concrete grammar

<gate definition> ::=
{ { <gate symbol 1> | <inherited gate symbol 1> }
is associated with { <gate> <signal list area> }***set***
| { <gate symbol 2> | <inherited gate symbol 2> }
is associated with { <gate> <signal list area> <signal list area> }***set***
} [***is connected to*** <endpoint constraint>]

<gate definition> is extended compared with Basic SDL-2010 to allow <inherited gate symbol 1> and <inherited gate symbol 2> for inherited gates and <endpoint constraint>.

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

<textual endpoint constraint> ::=
[**atleast**] <identifier>

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

<inherited gate symbol 2> ::=
◀-------------▶

An <inherited gate symbol 1> or <inherited gate symbol 2> shall only appear in a subtype definition, and it shall be used in the subtype for the gate with the same <gate name> specified in the supertype.

<signal list area>s and any <endpoint constraint> associated with an <inherited gate symbol 1> or <inherited gate symbol 2> are additions to those of the gate definition in the supertype.

The <identifier> of the <textual endpoint constraint> of an <endpoint constraint> with a <block symbol> (<process symbol>, <state symbol>) shall denote the definition of a block type (process type, state type, respectively). If <textual endpoint constraint> is specified for the gate in the supertype, the <identifier> of an (added) <textual endpoint constraint> shall denote the same type or a subtype of the type denoted in the <textual endpoint constraint> of the supertype.

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

If the type denoted by <base type> in a <typebased block definition> or <typebased process definition> contains channels, the following rule applies: for each combination of a gate, a remote procedure (or remote variable), 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 remote procedure or remote variable, respectively (or has no explicit <signal list area> associated if Recommendation [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

See [ITU-T Z.101].

8.3 Context parameters

In order for a type definition to be used in different contexts, both within the same system specification and within different system specifications, it is allowed to parameterize types with context parameters. Context parameters are replaced by actual context parameters as defined in clause 8.1.2.

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

Context parameters optionally have constraints (that is, required properties any entity denoted by the corresponding actual identifier shall have). The context parameters have these properties inside the type.

Concrete grammar

```
<formal context parameters> ::=
                        <context parameters start> <formal context parameter list> <context parameters end>

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

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

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

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

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

<formal context parameter> ::=
                        <agent type context parameter>
                   |    <agent context parameter>
                   |    <procedure context parameter>
                   |    <remote procedure context parameter>
                   |    <signal context parameter list>
                   |    <variable context parameter list>
                   |    <remotevariable context parameter list>
                   |    <timer context parameter list>
                   |    <synonym context parameter list>
                   |    <sort context parameter>
                   |    <compositestate type context parameter>
                   |    <gate context parameter>
                   |    <interface context parameter list>
```

The scope unit of a type definition with formal context parameters defines the names of the formal context parameters. These names are therefore visible in the definition of the type, and also in the definition of the formal context parameters.

An <actual context parameter> shall not be a <constant primary> unless it is for a synonym context parameter. A <constant primary> is a <primary> that is a valid <constant expression> (see clause 12.2.1 of [ITU-T Z.101]).

Formal context parameters are not allowed to be used as <base type> in <type expression> or in **atleast** constraints of <formal context parameters>.

Constraint is specified by a constraint specification, which is a <formal context parameter> with a constraint. A constraint specification introduces the entity of the formal context parameter followed by either a constraint signature or an **atleast** clause. A constraint signature introduces directly sufficient properties of the formal context parameter. An **atleast** clause denotes that the formal context parameter shall be replaced by an actual context parameter, which is the same type or a subtype of the type identified in the **atleast** clause. Identifiers following the keyword **atleast** in this clause shall identify type definitions of the entity kind of the context parameter and shall be neither formal context parameters nor parameterized types.

A formal context parameter of a type shall be bound only to an actual context parameter of the same entity kind that meets the constraint of the formal parameter.

A context parameter using other context parameters in its constraint shall not be bound before the other parameters are bound, and if this means there is no possible order for binding the context parameters, the model is not valid SDL-2010.

It is allowed to omit trailing commas in the <actual context parameter list>.

Model

The formal context parameters of a type definition that is neither a subtype definition nor defined by binding formal context parameters in a <type expression> are the parameters specified in the <formal context parameters>.

Context parameters of a type are bound in the definition of a <type expression> to actual context parameters. In this binding, occurrences of formal context parameters inside the parameterized type are replaced by the actual parameters. During this binding of identifiers contained in <formal context parameter>s to definitions (that is, deriving their qualifier; see clause 6.6 of [ITU-T Z.101]), local definitions other than the <formal context parameters>s are ignored.

Parameterized types shall not be used as actual context parameters.

If a scope unit contains <specialization>, any omitted actual context parameter in the <specialization> is replaced by the corresponding <formal context parameter> of the <base type> in the <type expression>, and this <formal context parameter> becomes a formal context parameter of the scope unit.

8.3.1 Agent type context parameter

Concrete grammar

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

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

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

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

8.3.2 Agent context parameter

Concrete grammar

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

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

<agent signature> ::=

```
                              <sort list>
                        |      [ <end> ] fpar <aggregation kind> <sort> {, <aggregation kind> <sort> }
```

NOTE – The two alternatives of <agent signature> are equivalent.

An actual agent parameter shall identify a block definition if **block** is specified in the <agent context parameter>; otherwise it shall identify a process definition.

In the case of **atleast** <agent type identifier> constraint, the actual agent type shall be the same as, or a subtype of, the constraint agent type with no addition of formal parameters to those of the constraint type.

In the case of a <colon> <agent type identifier> constraint, the actual agent type shall be the type denoted by <agent type identifier>.

In the case of an <agent signature> constraint, the actual agent shall be compatible with the <agent signature>. An agent definition is compatible with the <agent signature> if the formal parameters of the agent definition have the same aggregation kind and sorts as the corresponding elements of the <agent signature>, and both definitions have the same *Agent-kind*.

8.3.3 Procedure context parameter

Concrete grammar

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

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

<procedure signature in constraint> ::=
                              <procedure signature>
                        |     <legacy procedure signature>

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

<legacy procedure signature> ::=
                              [ <end> ]
                        {          [ returns <sort> ]
                              |    fpar <formal parameter> {, <formal parameter> }* [ <end> returns <sort> ] }
```

NOTE – The two alternatives of <procedure signature in constraint> are equivalent. The <sort> of <result> is equivalent to the <sort> after **returns**.

In the case of **atleast** <procedure identifier>, an actual procedure parameter shall identify a procedure definition that is the same as or specialization of the procedure of the constraint.

In the case of <procedure signature in constraint>, an actual procedure parameter shall identify a procedure definition compatible with the formal procedure signature given by <procedure signature in constraint>.

A procedure definition is compatible with the formal procedure signature if both have a result of the same <sort> or if neither returns a result, and:

a) the formal parameters of the procedure definition have the same sorts as the corresponding parameters of the signature each with the same <parameter kind>; or

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

8.3.4 Remote procedure context parameter

Concrete grammar

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

An actual parameter to a **remote** procedure context parameter shall identify a <remote procedure definition> compatible with formal procedure signature given by <procedure signature in constraint>.

8.3.5 Signal context parameter

Concrete grammar

```
<signal context parameter list> ::=
                              signal <signal context parameter name>
                                   { , <signal context parameter name>  }*

<signal context parameter name> ::=
                                   <signal name> [<signal constraint>]

<signal constraint> ::=
                              atleast <signal identifier> | <signal signature>

<signal signature> ::=
                              <sort list>
```

Each <signal name> is a signal context parameter in the context parameter list in the order given, optionally with the following <signal constraint>. The actual signal parameter shall identify a signal definition.

In the case of **atleast** <signal identifier>, an actual signal parameter shall identify a signal definition that is the same as, or a subtype of, the signal type of the constraint.

In the case of <signal signature>, an actual signal parameter shall identify a signal definition that is compatible with the formal <signal signature>. A signal definition is compatible with the formal <signal signature> if each parameter of the signal definition has the same aggregation kind and sort as the corresponding parameter of the <signal signature>.

8.3.6 Variable context parameter

Concrete grammar

```
<variable context parameter list> ::=
                              dcl <variable context parameter names>
                                   { , <variable context parameter names>  }*

<variable context parameter names> ::=
                              <variable name> { , <variable name>}* <variable constraint>

<variable constraint> ::=
                              <sort>
```

Each <variable name> is a variable context parameter in the context parameter list in the order given with the sort identified by the following <variable constraint>. An actual parameter shall be a variable, or a formal agent parameter or a formal procedure parameter of the same sort as the sort of the variable context parameter.

8.3.7 Remote variable context parameter

Concrete grammar

```
<remotevariable context parameter list> ::=
                               remote <remotevariable contextparameter names>
                                   { , <remotevariable contextparameter names> }*
```

<remotevariable contextparameter names> ::=
<remote variable name> { **,** <remote variable name>}* <variable constraint>

Each <remote variable name> is a remote variable context parameter in the context parameter list in the order given with the sort identified by the following <variable constraint>. An actual parameter shall identify a <remote variable definition> of the same sort as the sort of the remote variable context parameter.

8.3.8 Timer context parameter

Concrete grammar

<timer context parameter list> ::=
timer <timer context parameter name>
{ **,** <timer context parameter name> }*

<timer context parameter name> ::=
<timer name> [<timer constraint>]

<timer constraint> ::=
<sort list>

Each <timer name> is a timer context parameter in the context parameter list in the order given, optionally with a constraint given by the following <timer constraint>. An actual timer parameter shall identify a timer definition that is compatible with the formal sort constraint list. A timer definition is compatible with a formal sort constraint list if the sorts of the timer are the same sorts as in the sort constraint list.

8.3.9 Synonym context parameter

Concrete grammar

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

<synonym context parameter name> ::=
<synonym name> <synonym constraint>

<synonym constraint> ::=
<sort>

Each <synonym name> is a synonym context parameter in the context parameter list in the order given with the sort given by the following <synonym constraint>. An actual synonym parameter shall be a constant expression of the same sort as the sort of the constraint.

Model

If the actual parameter is a <constant expression> that is not a <synonym identifier>, there is an implied definition of an anonymous synonym in the context surrounding the type being defined with the context parameter.

8.3.10 Sort context parameter

Concrete grammar

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

In <sort context parameter> the keyword pair **value type** has the same meaning as the keyword **newtype**.

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

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

If <sort constraint> is omitted, any sort is allowed as the actual sort parameter.

In the case of a <sort constraint> that is **atleast** <sort>, an actual sort parameter shall be the sort given by <sort> or a subtype (without renaming; see [ITU-T Z.104]) of this sort.

In the case of a <sort constraint> that is <sort signature>, an actual sort parameter shall be compatible with the formal sort signature. A sort is compatible with the formal sort signature if the literals of the sort include the literals in the formal sort signature, and the operations defined by the data type that introduced the sort include the operations in the formal sort signature, and these operations have the same signatures.

A <literal signature> of the <sort signature> shall not contain <named number>.

8.3.11 Composite state type context parameter

Concrete grammar

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

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

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

An actual composite state type parameter shall identify a composite state type definition.

In the case of a <composite state type constraint> that is **atleast** <composite state type identifier>, the actual composite state type shall be the same or a subtype of the constraint composite state type <composite state type identifier>, with no addition of formal parameters to those of the constraint type.

In the case of a <composite state type constraint> that is a <composite state type signature>, the actual composite state type shall be compatible with the formal composite state type signature. A composite state type definition is compatible with the formal composite state type signature if each formal parameter to the composite state type definition has the same aggregation kind and sort as the corresponding parameter of the <composite state type constraint>.

8.3.12 Gate context parameter

Concrete grammar

<gate context parameter> ::=
gate <gate> <gate constraint>

<gate constraint> ::=
in [**from** <textual endpoint constraint>] [**with** <signal list>]
[**out** [**to** <textual endpoint constraint>] [**with** <signal list>]]
| [**out** [**to** <textual endpoint constraint>] [**with** <signal list>]
[**in** [**from** <textual endpoint constraint>] [**with** <signal list>]]]

out or **in** in a <gate constraint> denotes the direction of <signal list>, outward from or inward to the type respectively.

Where both **in** and **out** are specified in a <gate constraint>, this is for a bidirectional gate, and if two <textual endpoint constraint>s are given they shall be the same.

An actual gate parameter shall identify a gate definition. The outward signal list of the actual gate parameter shall contain all elements mentioned **with** <signal list> of the corresponding **out** formal gate context parameter. The **in** formal gate context parameter shall mention **with** <signal list> all inward elements of the actual gate parameter.

8.3.13 Interface context parameter

Concrete grammar

```
<interface context parameter list> ::=
                    interface <interface context parameter name>
                         { , <interface context parameter name> }*

<interface context parameter name> ::=
                         <interface name> [<interface constraint>]

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

Each <interface name> is an interface context parameter in the context parameter list in the order given, optionally with the constraint given by the following <interface constraint>. An actual interface parameter shall identify an interface definition.

In the case of an <interface constraint>, the type of the interface shall be the same or a subtype of the interface type identified by **atleast** <interface identifier>.

8.4 Specialization

It is allowed to define a type as a specialization of another type (the supertype), yielding a new subtype. A subtype optionally has properties in addition to the properties of the supertype, and it optionally redefines virtual local types and transitions. Except in the case of interfaces, there is at most one supertype.

Virtual types optionally have constraints (that is, properties any redefinition of the virtual type shall have). These properties are used to guarantee properties of any redefinition.

8.4.1 Adding properties

Concrete grammar

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

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

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

a) T shall not enclose subT;

b) T shall not be a specialization of subT;

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

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

The <type expression> of the <specialization> in:

a) <agent additional heading> represents the *Agent-type-identifier* of *Agent-type-definition* in clause 8.1.1.1 of [ITU-T Z.101];

b) <composite state type heading> or <state aggregation type heading> represents the *Composite-state-type-identifier* of *Composite-state-type-definition* in clause 8.1.1.5;

c) <procedure heading> represents the *Procedure-identifier* of *Procedure-definition* in clause 9.4 of [ITU-T Z.101].

The specialization of data types is defined in [ITU-T Z.104].

Semantics

The resulting content of a specialized type definition with local definitions consists of the content of the supertype followed by the content of the specialized definition. This implies that the set of definitions of the specialized definition is the union of those given in the specialized definition itself and those of the supertype. The resulting set of definitions shall obey the rules for distinct names as given in clause 6.6 of [ITU-T Z.101]. However, exceptions to this rule are:

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

b) a gate of the supertype is optionally given an extended definition (in terms of signals conveyed and endpoint constraints) in a subtype – this is specified by a gate definition of the same name as that of the supertype;

c) if the <type expression> contains an <actual context parameter list>, any occurrence of the <base type> of the <type expression> is replaced by the name of the subtype;

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

e) an operator or a method of the supertype is not inherited if an operator or method with a signature equal to the signature of the specialized operator or method is already present in the subtype.

The formal context parameters of a subtype are the unbound, formal context parameters of the supertype definition followed by the formal context parameters of the specialized type (see clause 8.2).

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

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

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

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

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

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

A specialized data type definition is allowed to add literals or fields to the inherited type constructors, it is allowed to add operators and methods, and it is allowed to add default initializations or default assignment (see [ITU-T Z.104]).

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

NOTE – When a gate in a subtype is an extension of a gate inherited from a supertype, the <inherited gate symbol 1> or <inherited gate symbol 2> is used in the concrete syntax.

8.4.2 Virtuality and virtual type

Within a type, some items (component types, transitions/saves – see clause 8.4.3; operation signatures – see clause 12.1.3 of [ITU-T Z.104]; and default initializations – see clause 12.3.3.2 of [ITU-T Z.104]) are marked as virtual, meaning that when the type is specialized it is allowed to redefine these items in the specialization. These kinds of component have a virtuality, which is virtual if redefinition is allowed, is redefined if redefined with further redefinition allowed, and is finalized if redefined with no further redefinition allowed.

The specification of an agent type, procedure or state type as a virtual type is allowed when it is defined locally to another type (denoted as the *enclosing* type). The redefinition of a virtual type (as **redefined** or **finalized**) is allowed in specializations of the enclosing type.

Concrete grammar

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

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

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

A virtual type is a type having **virtual** or **redefined** as <virtuality>. A type that has **finalized** as <virtuality>, or that does not have <virtuality> is not a virtual type. A redefined type is a type having **redefined** or **finalized** as <virtuality>. Redefinition is only allowed for virtual types. Every redefined type shall be directly or indirectly (via another redefined type) a redefinition of a virtual type that is not redefined (that is, with <virtuality> **virtual**).

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

A virtual type and its constraints shall not have context parameters.

Only virtual types are allowed to have <virtuality constraint> specified.

If <virtuality> is present in both a definition reference and the referenced definition, then they shall be the same. If <procedure preamble> is present in both a procedure reference and in the referenced procedure definition, they shall be the same.

A virtual agent type shall have exactly the same formal parameters, and at least the same gates and interfaces with at least the definitions as those of its constraint. A virtual state type shall have exactly the same formal parameters, and at least the same state connection points as its constraint. A virtual procedure shall have exactly the same formal parameters as its constraint.

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

In the case of an implicit constraint, redefinition involving **inherits** shall be a subtype of the constraint.

Accessing a virtual type by means of a qualifier denoting one of the supertypes implies, the application of the (re)definition of the virtual type given in the actual supertype denoted by the qualifier. A type T whose name is hidden in an enclosing subtype by a redefinition of T is made visible by qualification with a supertype name (that is, a type name in the inheritance chain). The qualifier consists of only one path item denoting the particular supertype.

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

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

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

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

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

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

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

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

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

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

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

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

Semantics

The redefinition of a virtual type is allowed in the definition of a subtype of the enclosing type of the virtual type. In the subtype, it is the definition from the subtype that defines the type of instances of the virtual type, also when applying the virtual type in parts of the subtype inherited from the supertype. A virtual type that is not redefined in a subtype definition has the definition as given in the supertype definition.

8.4.3 Virtual transition/save

Transitions or saves of a process type, state type or procedure are specified to be virtual transitions or saves by means of the keyword **virtual**. The redefinition of virtual transitions or saves is allowed in specializations. This is indicated by transitions or saves with the same state or signal, respectively, and with the keyword **redefined** or **finalized**.

Concrete grammar

The syntax of virtual transition and save is introduced in clause 9.4 (virtual procedure start), clause 10.5 (virtual remote procedure input and save), clause 11.1 (virtual process start), clause 11.3 (virtual input), clause 11.4 (virtual priority input), clause 11.5 (virtual continuous signal), clause 11.7 (virtual save), and clause 11.9 (virtual spontaneous transition).

A virtual transition (or save) is a transition (or save respectively) having **virtual** or **redefined** as <virtuality>. A transition (or save) that has **finalized** as <virtuality>, or that does not have <virtuality> is not a virtual transition (or save respectively). A redefined transition (or save) is a transition (or save respectively) having **redefined** or **finalized** as <virtuality>. Redefinition of a transition or save is only allowed for if the original is virtual. Every redefined transition (or save) shall be directly or indirectly (via another redefined transition/save) a redefinition of a virtual transition (or save respectively) that is not redefined (that is, with <virtuality> **virtual**).

A state shall not have more than one virtual spontaneous transition.

A redefinition of a transition (or save) marked with **redefined** in a specialization is allowed in further specializations, while a transition (or save) marked with **finalized** shall not be given new definitions in further specializations.

Redefinition of a virtual start transition to a redefined start transition is allowed.

It is allowed to redefine a virtual priority input or virtual input transition or a virtual save to a redefined priority input or redefined input transition or to a redefined save. The virtual item and the redefined item shall both have the same signal and if one mentions a gate the other shall mention the same gate.

It is allowed to redefine a virtual spontaneous transition a redefined spontaneous transition.

It is allowed to redefine a virtual continuous transition to a redefined continuous transition. The redefinition is indicated by the same priority (if present) as the virtual continuous transition. If several virtual continuous transitions exist in a state, then each of these shall have a distinct priority. If only one virtual continuous transition exists in a state, it is allowed to omit the priority.

It is allowed to redefine a virtual remote procedure input transition or virtual remote procedure save to a redefined remote procedure input transition or to a redefined remote procedure save. The virtual item and the redefined item shall both have the same procedure and if one mentions a gate the other shall mention the same gate.

Semantics

Redefinition of virtual transitions/saves corresponds closely to redefinition of virtual types (see clause 8.4.2).

In the subtype, for any particular state and stimulus it is the definition from the subtype that defines the virtual transition or save. For a virtual transition or save that is not redefined, the subtype definition has the definition as given in the supertype definition.

9 Agents

Concrete grammar

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

<agent text area> is extended compared with Basic SDL-2010 to allow <remote procedure definition>, <remote variable definition>, <macro definition> and <select definition> as alternatives in the <text symbol>.

<agent text area> is extended compared with Basic SDL-2010 to allow a <variable definition> in a system type or block type. Such variables are visible to enclosed agents, but are accessed as remote variables.

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

<state machine area> is extended compared to Basic SDL-2010 to include <inherited state machine>.

<inherited state machine> ::=
<dashed state symbol> ***contains*** { [<state name>]{ <gate>*}***set*** }

An <inherited state machine> shall only appear in a subtype definition, and is used in the subtype to contain each <gate> needed for channel connections in the same way as each <gate> in a <state machine area> with a <typebased composite state>. The <state name> (if present) shall be the same as the name of the state machine of the supertype.

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

<agent area> is extended compared to Basic SDL-2010 to include <inherited agent definition>.

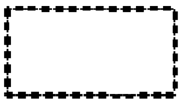
An <inherited agent definition> shall only appear in a subtype definition and shall be used in the subtype definition to represent the agent defined in the supertype. The <block name> or <process name> of the <inherited agent definition> shall represent the *Agent-name* of an inherited *Agent-definition* in the subtype. If a <number of instances> is given, this represents the *Number-of-instances* of the inherited *Agent-definition* in the subtype; otherwise the *Number-of-instances* is the same as the *Number-of-instances* of the inherited *Agent-definition* in the supertype. The *Agent-type-identifier* of the inherited *Agent-definition* in the subtype is the same as the *Agent-type-identifier* of the inherited *Agent-definition* in the supertype.

NOTE – It is allowed to specify additional channels connected to gates of an inherited agent and to change the number of instances.

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

<inherited block definition> ::=
<dashed block symbol> ***contains***
{ <block name> [<number of instances>] { <gate>* }***set*** }

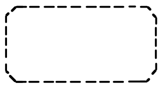
<dashed block symbol> ::=



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

<inherited process definition> ::=
<dashed process symbol> ***contains***
{ <process name> [<number of instances>]{ <gate>* }***set*** }

<dashed process symbol> ::=



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

Semantics

A variable defined in the <agent text area> of a system type or block type is visible to enclosed agents and is called a global variable. A global variable of a system or block is accessed through set and get remote procedure calls by enclosed agents. For the enclosing agent itself these global variables are local variables, and are created when the agent is created.

In addition to transitions taken from the state of an agent on signal consumption when waiting in a state, a spontaneous transition is also able to initiate a transition independent of any enabled signals being present in the input port.

9.1 System

System is as defined in [ITU-T Z.101].

9.2 Block

Model

A block with a global variable definition (a <variable definition> in the <agent text area> of the block type of the block) has a state machine that is interpreted concurrently with agents in the block. Access from contained agents in the block to a global variable of the block is covered by two implicitly defined remote procedures for setting and getting the data item associated with the variable. These procedures are provided by the state machine of the block.

A block type `bt` with global variables is transformed by moving the global variables from the <agent text area> of `bt` to a new (anonymously named) state type `st1` for `bt` that replaces the existing state type `st` for the state machine of `bt`. The state type `st1` has a <specialization> "**`inherits`** `st` **`adding`**", and adds each <variable definition> deleted from the <agent text area> of `bt` to a <composite state text area> of `st1`. For each variable `v` in `b`, `st1` has two exported procedures with anonymous implicit names, but here are called `set_v` (with an **in**-parameter of the sort of `v`) and `get_v` (with a return type being the sort of `v`). Each assignment to `v` from enclosed definitions of `bt` is transformed to a remote call of `set_v`. Each occurrence of `v` in expressions in enclosed definitions is transformed to a remote call of `get_v`. These occurrences also apply to occurrences in procedures defined in `bt`, as these are transformed into procedures local to the calling agents. There is no ambiguity between the remote procedure calls for different instances block type `bt`, because each instance has implicit remote procedure definitions for both procedures. The `set_v` and `get_v` procedures have a special property that means they are handled in the stopping condition of instances of `bt`.

This transformation takes place after replacing agent definitions with typebased agent definitions and transforming context parameters, and before the transforming of remote procedures.

NOTE – The `set_v` and `get_v` procedures have a parameter that holds the complete value associated with the variable, so that using these procedures for a global block variable with a significant size (such as an array, vector, string, or large structure) to write or read an element of the variable is probably inefficient compared with providing explicit remote procedures that write or read just that element.

9.3 Process

Process is as defined in [ITU-T Z.101].

NOTE – State aggregation has also alternating interpretation. However, alternating processes of a process each have their own input port and their own **self**, **parent**, **offspring** and **sender**. In the case of state aggregation there is only one input port and one set of **self**, **parent**, **offspring** and **sender** belonging to the container agent.

9.4 Procedure

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
[*Abstract*]

Procedure-definition is extended to optionally allow *Abstract* to be specified. See clause 8.1.3 for more information on *Abstract*.

Concrete grammar

```
<procedure heading> ::=
                    <procedure preamble>
                    procedure [<qualifier>] <procedure name>
                    [<formal context parameters>] [<virtuality constraint>]
                    [<specialization>]
                    [<procedure formal parameters>]
                    [<procedure result>]
```

<procedure heading> is extended compared with Basic SDL-2010 to allow <formal context parameters>, <virtuality constraint>, and <specialization> so that a procedure is able to inherit from another procedure.

```
<procedure preamble> ::=
                    <type preamble> [ <exported> ]

<exported> ::=
                    exported [ as <remote procedure identifier> ]
```

<procedure preamble> is extended compared with Basic SDL-2010 to allow procedures to be exported.

exported in a <procedure preamble> means that calling the procedure as a remote procedure is allowed, according to the model in clause 10.5.

An exported procedure shall not have formal context parameters and its enclosing scope shall be an agent type or a composite state type. If the enclosing scope is a composite state type, no composite state based on this state type shall be used directly or indirectly in the procedure.

If present, **exported** is inherited by any subtype of a procedure. A virtual exported procedure shall contain **exported** in all redefinitions. Virtual types including virtual procedures are described in clause 8.4.2. The optional **as** clause in a redefinition shall denote the same <remote procedure identifier> as in the supertype. If omitted in a redefinition, the <remote procedure identifier> of the supertype is implied.

Two exported procedures in an agent type or any enclosed composite state type of the agent type shall not mention the same <remote procedure identifier>.

The <remote procedure identifier> following **as** in an exported procedure definition shall denote a <remote procedure definition> with the same signature as the exported procedure. In an exported procedure definition with no **as** clause, there shall be a <remote procedure definition> in a surrounding scope with the same name and signature as the exported procedure and the nearest such <remote procedure definition> is used.

If **exported** is given in a procedure reference, the referenced procedure has to be an exported procedure and if a <remote procedure identifier> is also given, the procedure has to identify the same remote procedure definition.

```
<entity in procedure> ::=
                                <variable definition>
                        |       <data definition>
                        |       <select definition>
                        |       <macro definition>
```

Basic SDL-2010 <entity in procedure> is extended to allow <select definition> or a <macro definition>.

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

Basic SDL-2010 <procedure start area> is extended to allow <virtuality> for the start to be given. A <procedure start area> which contains <virtuality> of **virtual** or **redefined** is called a virtual procedure start. Virtual procedure start is further described in clause 8.4.3.

NOTE – If a subtype *Procedure-definition* is implicitly created locally in the enclosing agent of the *Call-node* or *Value-returning-call-node* (where otherwise the *Procedure-definition* is external to this agent), identifiers of items (such as variables) in the *Procedure-definition* are bound in the context of the super type *Procedure-definition* rather than the context of the *Call-node* or *Value-returning-call-node*.

10 Communication

10.1 Channel

Semantics

A remote procedure or remote variable on a channel is mentioned as outgoing from an importer and incoming to an exporter.

10.2 Connection

This feature is a shorthand notation associated with agent diagrams and is not included in Comprehensive SDL-2010.

10.3 Signal

Abstract grammar

Signal-definition	::	*Signal-name*
		*Signal-parameter**
		[*Signal-identifier*]
		[*Abstract*]

Signal-definition is extended compared with Basic SDL-2010 to include an optional *Signal-identifier* and optionally *Abstract*. The optional *Signal-identifier* of a *Signal-definition* identifies the base type (if any). See clause 8.1.3 for more information on *Abstract*.

Concrete grammar

```
<signal definition> ::=
                                <type preamble>
                                <signal name>
                                [<formal context parameters>]
                                [<virtuality constraint>]
                                [<specialization>]
                                [<sort list>]
```

<signal definition> is extended compared with Basic SDL-2010 to allow <formal context parameters>, <virtuality constraint>, and <specialization>, so that a signal is allowed context

parameters, is allowed redefinition in subtypes to be constrained, and is allowed to be a specialization of another signal.

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

Virtuality is explained in clause 8.4.2.

10.4 Signal list area

Concrete grammar

```
<signal list item> ::=
                        <signal identifier>
                    |   <timer identifier>
                    |   ( <interface identifier> )
                    |   procedure <remote procedure identifier>
                    |   remote <remote variable identifier>
```

<signal list item> is extended compared with Basic SDL-2010 to allow remote procedures and variables to be identified. The *Signal-identifier-**set*** of the *Channel-path* includes the identity of the implicit signals for each remote procedure and each remote variable.

The condition in clause 12.1.2 of Basic SDL-2010 is extended to include remote procedures and remote variables: 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> or a <remote procedure identifier> or <remote variable identifier>.

A <signal list item> of a <stimulus> (of an <input list> or <priority input list>) shall not denote a <remote variable identifier> and if it denotes a <remote procedure identifier> the <stimulus> parameters (including the parentheses) shall be omitted.

A <signal list item> of a <save item> shall not denote a <remote variable identifier> and if it denotes a <remote procedure identifier> or an <interface identifier>, the <stimulus> parameters (including the parentheses) shall be omitted.

10.5 Remote procedure

A client agent calls a procedure defined in another agent by a request to the server agent through a remote procedure call of a procedure in the server agent.

Concrete grammar

```
<remote procedure definition> ::=
                        remote procedure <remote procedure name>
                        <procedure signature> <end>
```

A <remote procedure definition> introduces the name and signature for imported and exported procedures. An exported procedure is a procedure with the keyword **exported**. An imported procedure is a procedure from another agent that is called via a remote procedure definition. The association between an imported procedure and an exported procedure is established by both referring to the same <remote procedure definition>.

```
<remote procedure call area> ::=
                        <procedure call symbol> contains <remote procedure call body>

<remote procedure call body> ::=
                        <remote procedure identifier> [<actual parameters>]
                        <communication constraints>
```

<communication constraints> is defined for output (see clause 11.13.4) and includes <timer communication constraint> defined below.

A remote procedure mentioned in a <remote procedure call body> shall be in the complete output set (see clause 8.1.1.1 and clause 9) of an enclosing agent type or agent set.

If a remote procedure is a value returning procedure, each action shall contain no more than one <remote procedure call body> used as an <expression0> for the same remote.

NOTE 1 – The constraint above on repeating calls of the same value returning remote more than once in an action is a consequence of the model, where the returned value is assigned to an implicit variable in the <remote procedure call body> transform inserted before the action (see below).

When the <communication constraints> is empty, there is a syntactic ambiguity between <remote procedure call body> and <procedure call body>. In this case, the contained <identifier> denotes a <procedure identifier>, if this is possible according to the visibility rules, and otherwise a <remote procedure identifier>.

In a <remote procedure call body>, a <communication constraints> list is associated with the last <remote procedure identifier>. For example, in

```
        call p to call q timer t via g
```

the **timer** t as well as **gate** g applies to the call of q.

<timer communication constraint> ::=
timer <timer identifier> [**(**[<variable>] { , [<variable>] }***)**]
[**connect** <connector name>]

The <timer identifier> of a <timer communication constraint> identifies the timer that is monitored for expiry. If the timer signal appears in the input port before the response to the remote communication, interpretation continues at the in-connector named by <connector name> according to the model below. The variables are used to receive the values (if any) of the timer signal.

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

Model

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

A remote procedure call body

```
        Proc(apar) to destination timer timeritem via viapath
```

is modelled by an exchange of implicitly defined signals. If the **to** or **via** clauses are omitted from the remote procedure call, they are also omitted in the following transformations. The communication uses the channels where the remote procedure has been mentioned in the <signal list> (the outgoing for the importer and the incoming for the exporter) of at least one gate or channel connected to the importer or exporter. The requesting agent sends a signal containing the actual parameters of the procedure call, except actual parameters corresponding to **out**-parameters, to the server agent and waits for the reply. In response to this signal, the server agent interprets the corresponding remote procedure, sends a signal back to the requesting agent with the results of all **in**/**out**-parameters and **out**-parameters (**in** parameters are excluded), and then interprets the transition for the remote procedure stimulus in the current state.

There are two anonymously named implicit <signal definition list> items for each <remote procedure definition> in a system definition. The <signal name> items in these <signal definition> items are denoted by `pCALL` and `pREPLY` respectively, where `p` is uniquely determined. The signals are defined in the same scope unit as the <remote procedure definition>. Both `pCALL` and `pREPLY` have a last parameter of the predefined `Integer` sort.

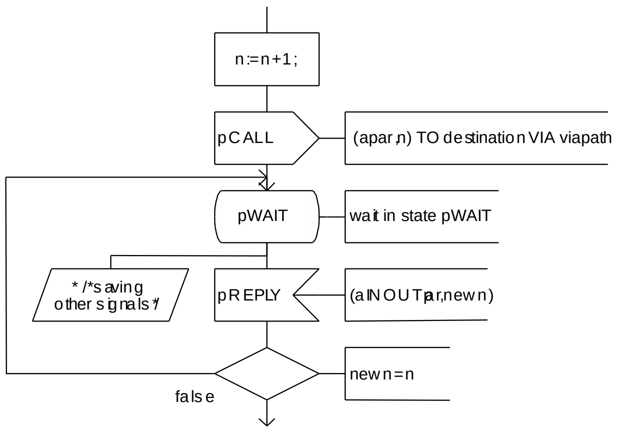
On each channel mentioning the remote procedure, the remote procedure is replaced by `pCALL`. For each such channel, if it is unidirectional the channel is made bidirectional. In the opposite direction this channel carries the signal `pREPLY`. The new channel has the same delaying property as the original one.

a) For each imported procedure, two implicit anonymous `Integer` variables (in this description called `n` and `newn`) are defined in the enclosing scope unit of the <remote procedure call body>, and `n` is initialized to 0. The same two variables (`n` and `newn`) are used for every <remote procedure call body> in the scope unit for the same remote procedure.

NOTE 2 – The parameter `n` is introduced to recognize and discard reply signals of remote procedure calls that were left through associated timer expiry.

If remote procedure is a value returning procedure, there is an implicit anonymous variable (in this description called `res`) defined in the enclosing scope unit of the <remote procedure call body> with the sort returned by the procedure.

The <remote procedure call body> is transformed as below, so that the following is inserted before the action that contained the <remote procedure call body>, where in the output the **to** clause is omitted if the destination is not present, and the **via** clause is omitted if it is not present in the original expression:



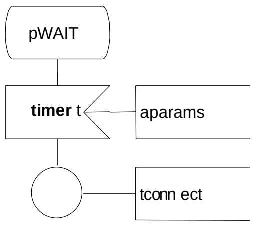
where

`apar` is the list of actual parameters except actual parameters corresponding to **out**-parameters, and `aINOUTpar` is the modified list of actual **in/out**-parameters and **out**-parameters, including the implicit variable `res` as an additional parameter if a value returning remote procedure call is transformed.

The transform is labelled with the label on the action containing the remote procedure call or a new label if this action is not labelled, and the preceding path is changed to join this label.

If a value returning remote procedure call is transformed, the true path above is terminated with a join to the action that contained the remote procedure call with a new label, and the remote procedure call is replaced by an access of the implicit variable `res` used to receive the returned value. Otherwise the remote procedure call action is removed, and the true path above is joined to the action following the remote procedure call action.

Additionally, the following is inserted if a <timer communication constraint> is included in <communication constraints>:



where

`t` is the <timer identifier> in the <timer communication constraint>;

`aparams` is the optional list of optional <variable> items given after the <timer identifier> in the <timer communication constraint>;

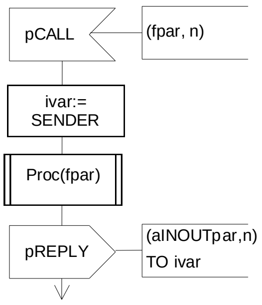
`tconnect` is the <connector name> if one is given in the <timer communication constraint>; otherwise `tconnect` is the name of the timer.

In all other states, `pREPLY` is discarded. This is not explicitly modelled: instead the handling of `pREPLY` is left unspecified in the transformed concrete syntax except for the `pWAIT` states with the consequence that there is an implicit transition (see clause 11.8 of [ITU-T Z.103]) for other states that discards the signal.

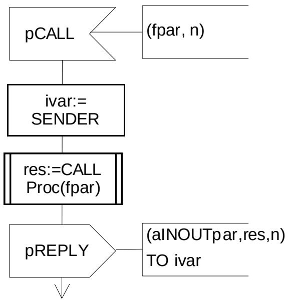
If the <remote procedure call body> was directly enclosed by a <remote procedure call>, it is an <action> that is the <remote procedure call> and the transform replaces the <remote procedure call body>. Otherwise the <remote procedure call body> is a <value returning procedure call> as an <expression0>, and transform is inserted before the action that contained the <value returning procedure call>, and the <value returning procedure call> is replaced in this action by an access of the implicit variable used to receive the returned value.

b) In the server agent, an implicit anonymous `Integer` variable (in this description called `n`) is defined for each <input area> that is a remote-procedure input. Furthermore, there is an implicit anonymous `Pid` variable (in this description called `ivar`) for each such <input area> defined in the scope where the remote procedure input occurs. If a value returning remote procedure call is transformed, an implicit anonymous variable (in this description called `res`) with the same sort as <sort> in <procedure result> is defined.

To all <state area>s with a remote procedure input transition, the following <input area> replaces the remote procedure input and leads to the transition for the remote procedure:



or,



if a value returning remote procedure call was transformed.

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

To all other <state area>s (excluding implicit states derived from input) where the remote procedure is not shown, the <input area> described above is added and terminates in a next state that returns to the same state.

NOTE 3 – There is a possibility of deadlock using the remote procedure construct, especially if no <destination> is given, or if <destination> does not denote a <pid expression> of an agent which is guaranteed by the specification to exist at the time of receiving the `pCALL` signal. Associated timers allow the deadlock to be avoided.

10.6 Remote variable

In SDL, a variable is always owned by, and local to, an agent instance. Normally, the variable is visible only to the agent instance that owns it and to the contained agents. If an agent instance in another agent needs to access the data associated with a variable, a signal interchange with the agent instance owning the variable is needed.

This signal exchanged is provided by the following shorthand notation, called imported and exported variables. The shorthand notation is also used to export data items to other agent instances within the same agent instance set.

Concrete grammar

The agent instance that owns a variable whose data items are exported to other agent instances is called the exporter of the variable. Other agent instances that use these data items are known as importers of the variable. The variable is called exported variable.

<remote variable definition> ::=
remote <remote variables of sort> {**,**<remote variables of sort>}* <end>

<remote variables of sort> ::=
<remote variable name> {**,**<remote variable name>}* <sort> [**nodelay**]

A <remote variable definition> introduces the name and sort for imported and exported variables. An exported variable definition is a variable definition with the keyword **exported**. The association between an imported variable and an exported variable is established by both referring to the same <remote variable definition>.

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

<import expression> ::=
import (<remote variable identifier> <communication constraints> **)**

If <destination> in the <communication constraints> of an <import expression> is an <agent identifier> or **this**, the <remote variable identifier> shall represent a remote variable contained in the interface of the agent type.

```
<export body> ::=
                        ( <variable identifier> { , <variable identifier> }* )
```

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

Each <action> shall contain no more than one <import expression> for the same remote variable.

Model

An agent instance is allowed to be both importer and exporter of the same remote variable. An overview of the two operations (a detailed model of which is given below) is:

a) Export operation

Exported variables have the keyword **exported** in their <variable definition>s, and have an implicit copy to be used in import operations.

An export operation is the interpretation of an <export body> by which an exporter discloses the current result of an exported variable. An export operation causes the storing of the current result of the exported variable into its implicit copy.

b) Import operation

An import operation is the interpretation of an <import expression> by which an importer accesses the result of an exported variable. The result is stored in an implicit variable denoted by the <remote variable identifier> in the <import expression>. The exporter containing the exported variable is specified by the <destination> in the <import expression>. If no <destination> is specified, then the import is from an arbitrary agent instance exporting the same remote variable. The association between the exported variable in the exporter and the implicit variable in the importer is specified by referring to the same remote variable in the export variable definition and in the <import expression>.

The import access is modelled by exchange of implicitly defined signals. The importer sends a signal to the exporter, and waits for the reply. In response to this signal, the exporter sends a signal back to the importer with the result contained in the implicit copy of the exported variable.

There are two implicit signal definitions for each variable of a <remote variable definition> in a system definition. A <remote variable definition> that defines multiple variables is expanded to a <remote variable definition> list with one variable per <remote variable definition>. The <signal name>s in the implicit signal definitions are denoted in this model by `xQUERY` and `xREPLY` respectively, where `x` denotes an implicit <name> associated with the <remote variable definition>. The signals are defined in the same scope unit as the <remote variable definition>. The signal `xQUERY` has an argument of the predefined sort Integer and `xREPLY` has arguments of the sort of the variable and Integer.

On each channel mentioning the remote variable, the remote variable is replaced by `xQUERY`. For each such channel, if it is unidirectional the channel is made bidirectional. In the opposite direction this channel carries the signal `xREPLY`.

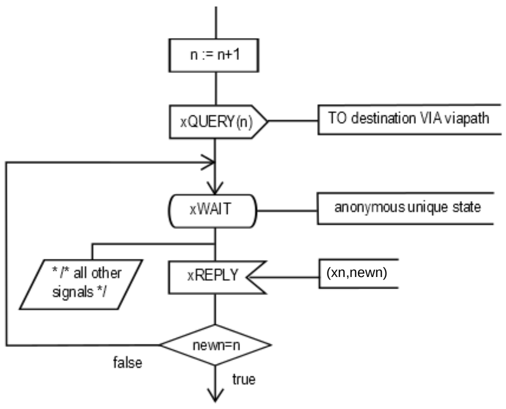
a) Importer

For each imported variable, two implicit anonymous `Integer` variables (in this description called `n` and `newn`) are defined in the enclosing scope unit of the <import expression>, and `n` is initialized to 0. The same two variables (`n` and `newn`) are used for every <import expression> in the scope unit for the same remote variable. In addition, an implicit anonymous variable (in this description called `xn`) of the sort of the remote variable is defined for each <import expression>.

The <import expression>

```
    import (x to destination via viapath)
```

is transformed so that the following is inserted before the action that contained the <import expression>, where in the output the **to** clause is omitted if the destination is not present, and the **via** clause is omitted if it is not present in the original expression:



– The insertion is labelled with the label on the action containing the import expression or a new label if this action is not labelled, and the preceding path is changed to join this label.

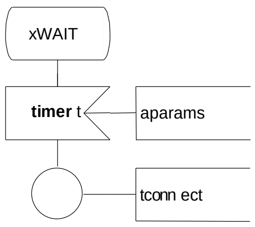
– The true path above is terminated with a join to the action that contained the import expression with a new label.

– The import expression is changed to an access of the variable `xn`.

In all other states, `xREPLY` is discarded. This is not explicitly modelled: instead the handling of `xREPLY` is left unspecified in the transformed concrete syntax except for the `xWAIT` states with the consequence that there is an implicit transition (see clause 11.8 of [ITU-T Z.103]) for other states that discards the signal.

NOTE 1 – Until 2017 `xREPLY` was saved in other states: the change to discarded is consistent with remote procedures.

Additionally, the following is inserted if a <timer communication constraint> is included in <communication constraints>:



where

`t` is the <timer identifier> in the <timer communication constraint>;

`aparams` is the optional list of optional <variable> items given after the <timer identifier> in the <timer communication constraint>;

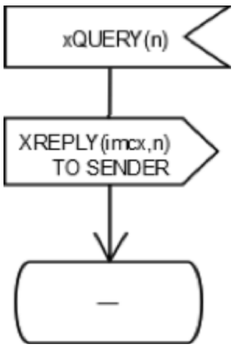
`tconnect` is the <connector name> if one is given in the <timer communication constraint>; otherwise `tconnect` is the name of the timer.

b) Exporter

For each exported variable, an implicit anonymous variable (in this description denoted by `imcx`) is defined to hold the exported value of the exported variable, and an implicit anonymous variable of type Integer (in this description denoted by `n`) is defined.

If a default initialization is attached to the export variable or if the export variable is initialized when it is defined, then the implicit copy `imcx` is also initialized, with the same result as the export variable.

To all <state area>s of the exporter, the following <input area> is added:



The <export statement>

```
    export x;
```

is transformed to the following:

```
    imcx := x;
```

NOTE 2 – There is a possibility of deadlock using the import construct, especially if no <destination> is given, or if <destination> does not denote a <pid expression> of an agent which is guaranteed by the specification to exist at the time of receiving the `xQUERY` signal. Specifying a set timer in the <timer communication constraint> avoids such a deadlock.

The keyword **nodelay** has no SDL-2010 or SDL-2000 meaning, though to be compatible with SDL-92 the channel conveying the signals for the remote variable should be a channel without delay.

11 Behaviour

11.1 Start

Abstract grammar

Named-start-node is introduced to define named state entry points of composite states.

Named-start-node	::	*State-entry-point-name* *Transition*

Each *Named-start-node* shall be in a *State-transition-graph* of a *Composite-state-graph*.

State-entry-point-name	=	*Name*

Concrete grammar

```
<start area> ::=
        <start symbol> contains { [<virtuality>] [<state entry point name>]}
        is followed by <transition area>
```

<start area> is extended compared with Basic SDL-2010 to allow <virtuality> so that the start is optionally virtual, and is also extended with a <state entry point name> for a named state entry point of a composite state. If <state entry point name> is given in a <start area>, the <start area> shall be the <start area> of a <composite state type diagram> and defines a *Named-start-node*.

NOTE 1 – A <start area> which contains <virtuality> is called a virtual start. Virtual start is further described in clause 8.4.3.

NOTE 2 – The grammar for a <composite state type diagram> containing <start area> items is defined in clause 11.11.1.

11.2 State

Abstract grammar

```
State-node          ::   State-name
                         Save-signalset
                         Input-node-set
                         {   Spontaneous-transition-set Continuous-signal-set
                         |   Composite-state-type-identifier Connect-node-set }
                         [ State-timer ]

State-timer         ::   Time-expression
                         Timer-identifier
                         Expression*
                         Transition
```

State-node is extended compared with Basic SDL-2010 to allow spontaneous transitions, continuous signals and state timers.

Concrete grammar

```
<state area> ::=
          <state symbol> contains <state list>
          is associated with
          {       <input association area>
             |    <priority input association area>
             |    <continuous signal association area>
             |    <spontaneous association area>
             |    <save association area>
             |    <connect association area> }*
          is associated with [ <state timer association area> ]
```

<state area> is extended compared with Basic SDL-2010 to include priority inputs, spontaneous transitions and continuous signals.

NOTE 1 – Although the concrete grammar allows <continuous signal association area> and <spontaneous association area> for a <state area> with a <state list item> that is a <typebased composite state> or <composite state list item>, for a composite state in the abstract grammar *Spontaneous-transition* and *Continuous-signal* are not allowed, therefore they are only valid for basic state items.

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

A <spontaneous transition area> represents a *Spontaneous-transition* of a *State-node*.

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

A <state timer association area> represents a *State-timer* of a *State-node*.

```
<state timer area> ::=
          <plain input symbol> contains { <virtuality> <state timer> }
          is followed by <transition area>

<state timer> ::=
          state timer <Time expression> | set <set clause>
```

When two <state> items contain the same <state name>, a <state timer> shall not be specified for both <state> items or both <state> items shall specify the same <state timer>.

NOTE 2 – A <state timer> with **state timer** is a *Model* (see below).

A <set clause> in a <state timer> does not represent a *Set-node* but represents a *State-timer* with the *Time-expression*, *Timer-identifier* and *Expression* list being represented in the same way (and with the same constraints) as these items of a *Set-node*.

Semantics

The interpretation is extended compared with Basic SDL-2010 to include the interpretation of *Input-node* items with a *Provided-expression*, the *Spontaneous-transition-**set***, the *Continuous-signal-**set*** and the optional *State-timer*.

When a state with a *State-timer* is entered, if the identified timer is inactive, it is set using the *Time-expression*, *Timer-identifier* and *Expression* list in the same way as setting a timer in a *Set-node*. The timer is reset making it inactive whenever a *Transition* of the *State-node* is interpreted (from an *Input-node*, or a *Continuous-signal*, or a *Spontaneous-transition* or a *Connect-node*), except if the *Transition* consists of just a *Terminator* that is a *Nextstate-node* that identifies the *State-node* for the *State-timer*. In this case, the timer remains active and therefore not set as the state is re-entered.

For each basic state, the *Save-signalset*, *Continuous-signal-**set*** and *Input-node-**set*** are interpreted in the following steps extended from Basic SDL-2010. 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. The *Spontaneous-transition-**set*** items are interpreted at any time as described below.

a) *State-timer* signals and priority inputs are handled:
   1) if a timer signal corresponding to a *State-timer* is in the input port, this is handled as described below and the step sequence is terminated; otherwise
   2) signals that have priority are handled in priority order (see clause 11.4); otherwise

b) in the order of the signals on the input port:
   1) it is evaluated if the current signal is enabled: the signal is not enabled if it is saved for the current state, or if it is saved for its gate of arrival for the current state, or if for every *Input-node* corresponding to the current signal and its gate of arrival there is a *Provided-expression* that interprets as false (these are interpreted in a arbitrary order);
   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, in *Priority-name* order (lowest first) of the *Continuous-signal-**set*** items, if any, items with equal *Priority-name* values being considered in an arbitrary order and no *Priority-name* value being treated as the highest priority (higher than any given *Priority-name* of a *Continuous-signal* of the current state):
   1) the *Continuous-expression* contained in the current *Continuous-signal* is interpreted;
   2) if the current continuous signal is enabled, the *Transition* of the continuous signal is interpreted (see clause 11.5); otherwise
   3) the next continuous signal is selected.

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, or if there is an *Input-node* with a *Provided-expression* that could have changed, or *Continuous-expression* that could have changed, the steps are repeated. A *Provided-expression* or *Continuous-expression* is able to change only if it contains a *Now-expression*, *Timer-active-expression*, or *Variable-access* to a variable defined in an enclosing agent that is changed by assignment in another agent instance or another state partition.

While the system remains in a state, a timer signal that corresponds to a *State-timer* of the state is put into the input port as soon as the system time reaches this Time value associated with the timer. The next time step (a) is entered within the state, the timer signal is consumed and interpretation proceeds from the state to the *Transition* of the *State-timer*.

When the *Transition* of an *Input-node*, or a *Continuous-signal*, or a *Spontaneous-transition* or *Connect-node* of a *State-node* is interpreted, the timer signal that corresponds to a *State-timer* of the *State-node* is reset in the same way as resetting a timer in a *Reset-node*, except in the case the *Graph-node* list of the *Transition* is empty and the *Terminator* is a *Nextstate-node* with *State-name* of the *State-node*. That is, the timer is reset, except if the transition leads back to the same state with no action.

At any time in a state for *State-node* with a *Spontaneous-transition* without a *Provided-expression*, the state machine is able to interpret the *Transition* of the *Spontaneous-transition* (see clause 11.9). Similarly, at any time in a state for *State-node* with *Spontaneous-transition* that has a *Provided-expression*, the state machine is able to interpret the *Provided-expression* of a *Spontaneous-transition* and subsequently, if the *Spontaneous-transition* was enabled, the *Transition* of the *Spontaneous-transition* (see clause 11.9).

Model

A <state timer> with **state timer** <Time expression> is equivalent to <state timer> with a <set clause> with a parameterless timer definition with an anonymous name set to the <Time expression>.

11.3 Input

Abstract grammar

Input-node	::	[*Priority-name*] *Signal-identifier* [*Gate-identifier*] [*Provided-expression*] [*Variable-identifier*]* *Transition*
Priority-name	=	*Nat*

Input-node is extended compared with Basic SDL-2010 to allow priority to be given for an input and a *Provided-expression* that has to be true for the signal to be enabled.

Concrete grammar

```
<input area> ::=
        <input symbol> contains { [<virtuality>] <input list> }
        {        is connected to <enabling condition association area>
            |    is followed by <transition area> }
```

<input area> is extended compared with Basic SDL-2010 to allow the input to be virtual and enabling conditions.

NOTE – As in Basic SDL-2010, an <input list> in Comprehensive SDL-2010 contains only one <stimulus> and shall not denote an <interface identifier>.

The <enabling condition association area> defines the <transition area> in the case of an enabling condition.

In the *Abstract grammar*, the <remote procedure identifier>s are also represented as *Signal-identifier*s.

An <input area> that contains <virtuality> is called a virtual input transition. Virtual input transition is further described in clause 8.4.3.

Semantics

The cases covered are extended compared with Basic SDL-2010 to include a *Provided-expression* in an *Input-node*.

If an *Input-node* has a *Provided-expression*, a signal instance is enabled for the *Input-node* only if the *Provided-expression* evaluates to true, and 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 if the current time is greater than or equal to the availability time for the signal instance. If the *Provided-expression* is false, the signal instance is not enabled by this *Input-node*.

11.4 Priority Input

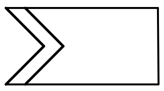
In some cases, it is convenient to express that reception of a signal takes priority over reception of other signals. This ordering is expressed by means of priority input.

Concrete grammar

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

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

<priority input symbol> ::=
```



```
<priority input list> ::=
                              <priority stimulus>

<priority stimulus> ::=
                              <stimulus> <priority clause>

<priority clause> ::=
                              priority <priority name>

<priority name> ::=
                              <integer name>
```

A <priority input association area> represents an *Input-node* with a *Priority-name*.

A <priority input area> which contains <virtuality> is called a virtual priority input. Virtual priority input is further described in clause 8.4.3.

Semantics

If an *Input-node* of a state has a *Priority-name*, the input is a priority input. An enabled signal in the input port with the highest priority (lowest *Priority-name* value) is consumed before any other signals are consumed. In step (a) of clause 11.2, the *Input-node-**set*** of the state is considered in order of the *Priority-name* values (lowest first, and those with the same *Priority-name* value in arbitrary order).

a) For each *Input-node* with the current priority value, in the order of the signals on the input port, if the current signal has the *Signal-identifier* of the *Input-node* and arrived at the gate identified by the *Gate-identifier* of the *Input-node* and (if there is one) the *Provided-expression* is true, the signal is enabled;

b) If the current signal is enabled, step (a) of clause 11.2 is complete and the signal is consumed for the *Input-node,* and these steps and the steps in clause 11.2 are also terminated; otherwise

c) If no enabled signal was found for the current priority value, the steps (a) and (b) here are repeated for the next higher *Priority-name* value (next lower priority), or if steps (a) and (b) have already been taken for the highest *Priority-name* value, the steps (b) to (d) of 11.2 are applied.

Other than the selection of the signal to consume as above, an *Input-node* with a *Priority-name* is the same as an *Input-node* without a *Priority-name*.

11.5 Continuous signal

A continuous signal is for the situations that arise where a transition should be interpreted when a certain condition is fulfilled. A continuous signal interprets a Boolean expression and the associated transition is interpreted when the expression returns the predefined Boolean value true.

Abstract grammar

Continuous-signal	::	*Continuous-expression* [*Priority-name*] *Transition*
Continuous-expression	=	*Boolean-expression*

Concrete grammar

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

<continuous signal area> ::=
                    <enabling condition symbol>
                    contains {
                            [<virtuality>] <continuous expression>
                            [ [<end>] priority <priority name> ] }
                    is followed by <transition area>

<continuous expression> ::=
                    <Boolean expression>
```

Semantics

The *Continuous-expression* is interpreted as part of the state to which its *Continuous-signal* is associated (see clause 11.2). If the *Continuous-expression* returns the predefined Boolean value true, the continuous signal is enabled.

The *Priority-name* determines the order of interpretation of the *Continuous-signal-**set*** items of a State-node (see clause11.2).

A <continuous signal area> that contains <virtuality> is called a virtual continuous signal. Virtual continuous transition is further described in clause 8.4.3.

11.6 Enabling condition

An enabling condition makes it possible to impose an additional condition on the consumption of a signal, beyond its reception, or to impose a condition on a spontaneous transition.

Abstract grammar

Provided-expression	=	*Boolean-expression*

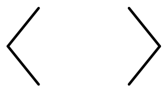
Concrete grammar

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

<enabling condition area> ::=
                              <enabling condition symbol> contains <provided expression>
                              is followed by <transition area>
```

The <transition area> corresponds to the *Transition* of the *Input-node* or *Spontaneous-transition* for the *Provided-expression*. The syntax appears here to get the correct graphical production of a flow line from the <enabling condition symbol> to the transition.

```
<enabling condition symbol> ::=
```



```
<provided expression> ::=
                              <Boolean expression>
```

The *Provided-expression* is represented by <provided expression>.

Semantics

The *Provided-expression* of an *Input-node* is interpreted as part of the state this *Input-node* is attached to (see clause 11.2).

A signal in the input port is enabled if the *Provided-expression* of an *Input-node* for the signal returns the predefined Boolean value true, or if the *Input-node* does not have a *Provided-expression*. The *Provided-expression* of a *Spontaneous-transition* is optionally interpreted at any time while the agent is in the state.

11.7 Save

Concrete grammar

```
<save area> ::=
                              <save symbol> contains { [<virtuality>] <save list> }
```

<save area> is extended compared with Basic SDL-2010 to include <virtuality>. A <save area> which contains <virtuality> is called a virtual save. Virtual save is further described in clause 8.4.3.

11.8 Empty clause

This clause is intentionally left blank.

11.9 Spontaneous transition

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

Abstract grammar

```
Spontaneous-transition          ::      [Provided-expression]
                                        Transition
```

Concrete grammar

```
<spontaneous transition area> ::=
                              <input symbol> contains { [<virtuality>] <spontaneous designator> }
                              {            is connected to <enabling condition association area>
                                    |      is followed by <transition area> }
```

The <enabling condition association area> defines the <transition area> in the case of an enabling condition.

<spontaneous designator> ::=
none

Semantics

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

After activation of a spontaneous transition, the **sender** expression of the agent returns **self**.

A <spontaneous transition area> that contains <virtuality> is called a virtual spontaneous transition. Virtual spontaneous transition is further described in clause 8.4.3.

11.10 Label (connector name)

In addition to Basic SDL-2010, the term body includes <state aggregation body area>, therefore all the <connector name>s defined in a body shall be distinct.

It is permissible to have a join from the body of the specialization to a connector defined in the supertype.

11.11 State machine and composite state

Basic SDL-2010 is extended to include named start nodes and exit points, entry and exit procedures, and aggregate state types.

11.11.1 Composite state graph

Abstract grammar

Composite-state-graph	::	*State-transition-graph* [*Entry-procedure-definition*] [*Exit-procedure-definition*]
State-transition-graph	::	[*State-start-node*] *Named-start-node-**set*** *State-node-**set*** *Free-action-**set***

Compared with Basic SDL-2010 *Composite-state-graph* is extended to include *Entry-procedure-definition* and *Exit-procedure-definition*; and *State-transition-graph* is extended to include a *Named-start-node-**set**.*

Entry-procedure-definition	=	*Procedure-definition*
Exit-procedure-definition	=	*Procedure-definition*

In the following, a procedure with states is a procedure that contains a state (explicit or implicit) or calls a procedure with states.

Entry-procedure-definition of a *Composite-state-graph* or *State-aggregation-node* is a procedure without parameters explicitly defined in the *Composite-state-graph* or *State-aggregation-node,* respectively, with the name entry. An entry procedure shall not be a procedure with states.

Exit-procedure-definition of a *Composite-state-graph* or *State-aggregation-node* is a procedure without parameters explicitly defined in the *Composite-state-graph* or *State-aggregation-node,* respectively, with the name exit. An exit procedure shall not be a procedure with states.

Concrete grammar

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

An <aggregation structure area> is similar to a <composite state structure area> except it contains a <state aggregation body area> rather than a <composite state body area>.

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

<composite state text area> is extended compared with Basic SDL-2010 to allow <macro definition> items and <select definition> items.

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

Basic SDL-2010 <composite state body area> is extended to allow more than one <start area> so that labelled start areas (a <start area> with a <state entry point name>) as well as the unlabelled start area are allowed.

At most, one of the <start area>s shall be unlabelled. Each additional labelled entry and exit point shall be defined by a corresponding <state connection point area>. Each additional labelled <start area> shall contain a different <state entry point name>.

A <start area> with a <state entry point name> (a labelled start) in a <composite state body area> shall refer only to <state entry point>s of the <composite state type diagram> directly enclosing the <composite state body area>. A <return area> with a <state exit point> (a labelled return) in a <composite state body area> shall refer only to <state exit point>s in the <composite state type diagram> directly enclosing the <composite state body area>.

If a <composite state body area> contains at least one <state area>, a <start area> shall be present.

<variable definition> in a <composite state text area> shall not contain **exported** <variable name>s, if the <composite state type diagram> is enclosed by a <procedure diagram>.

Semantics

The unlabelled *State-start-node* of the *Composite-state-graph* is interpreted when the *Nextstate-node* has no *State-entry-point-name*. A *Named-start-node* is interpreted as an additional entry point of the composite state. The *State-entry-point-name* of the *Nextstate-parameters* of a *Nextstate-node* defines which named start transition is interpreted.

A *Named-return-node* is an additional exit point of the composite state that is named (see clause 11.12.2.4).

Entry-procedure-definition and *Exit-procedure-definition*, if defined, are called implicitly when the state is entered and exited, respectively. It is not mandatory to define either or both procedures. The entry procedure is called before the start transition is invoked, or if the state is re-entered as a result of interpreting a *Nextstate-node* with **HISTORY**. The exit procedure is invoked after a *Return-node* of the *Composite-state-graph* is interpreted and before a transition attached directly to the *State-node* is interpreted, if there are such transitions. When an exception is raised in a composite state, the exit procedure is not invoked.

11.11.2 State aggregation

A state aggregation is a partitioning of a composite state. It consists of multiple composite states, which have an interpretation of alternating transitions. At any given time, each partition of a state aggregation is in one of the states of that partition, or (for one of the partitions only) in a transition, or has completed and is waiting for other partitions to complete. Each transition runs to completion.

Abstract grammar

State-aggregation-node	::	*State-partition-**set*** [*Entry-procedure-definition*] [*Exit-procedure-definition*]
State-partition	::	*Name* *Composite-state-type-identifier* *Connection-definition-**set***
Connection-definition	::	*Entry-connection-definition* | *Exit-connection-definition*
Entry-connection-definition	::	*Outer-entry-point Inner-entry-point*
Outer-entry-point	::	*State-entry-point-name*
Inner-entry-point	::	*Nextstate-parameters*
Exit-connection-definition	::	*Outer-exit-point Inner-exit-point*
Outer-exit-point	::	*State-exit-point-name*
Inner-exit-point	::	*State-exit-point-name*

The *State-entry-point-name* in the *Outer-entry-point* shall denote a *State-entry-point-definition* of the *Composite-state-type-definition* where the *State-aggregation-node* occurs. The *State-entry-point-name* of the *Inner-entry-point* shall denote a *State-entry-point-definition* of the composite state in the *State-partition*. The *State-exit-point-name* in the *Outer-exit-point* shall denote a *State-exit-point-definition* of the *Composite-state-type-definition* where the *State-aggregation-node* occurs. The *State-exit-point-name* of the *Inner-exit-point* shall denote a *State-exit-point-definition* of the composite state in the *State-partition*. The *State-entry-point-name* for the unlabelled entry point of a *Composite-state-type-definition* is a unique anonymous *Name*. The *State-exit-point-name* for the unlabelled exit point of a *Composite-state-type-definition* is unique anonymous *Name*.

For each *State-partition*, each of the entry points of the container state shall appear in exactly one *Connection-definition*. For each *State-partition*, each of the exit points of the *State-partition* shall appear in exactly one *Connection-definition*.

The input signal sets of the *State-partition* items within a composite state shall be disjoint. The input signal set of a *State-partition* is defined as the union of all signals appearing in an *Input-node* or the *Save-signalset* inside the composite state type, including nested states, and procedures mentioned in any *Call-node*.

Concrete grammar

<state aggregation body area> ::=
{ { <state partition area> | <state partition connection area>}* }***set***

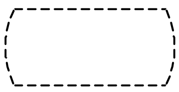
<state partition area> ::=
<typebased state partition definition>
| <inherited state partition definition>

<typebased state partition definition> ::=
<state symbol> ***contains*** { <typebased state partition heading> }

<typebased state partition heading> ::=
<state name> <colon> <composite state type expression>

<inherited state partition definition> ::=
<dashed state symbol> ***contains*** { <composite state identifier> }

<dashed state symbol> ::=



An <inherited state partition definition> shall only appear in a state aggregation subtype definition. The <composite state identifier> of a <inherited state partition definition> shall identify a state partition that is defined in the state aggregation supertype.

The <state partition area> and the <state partition connection area> items that have a <solid association symbol> attached to the <state partition area> represent a *State-partition*. If the <state partition area> is a <typebased state partition definition>, the <state name> and <composite state type expression> represent the *Name* and *Composite-state-type-identifier* of the *State-partition*. If the <state partition area> is an <inherited state partition definition>, the name and type of the state partition identified by the <composite state identifier> represent the *Name* and *Composite-state-type-identifier* of the *State-partition*.

```
<state partition connection area> ::=
                              <solid association symbol>
                              is attached to <frame symbol>
                              is attached to <state partition area>
                              {          is connected to { <outer entry points>}
                                              is connected to { <inner entry point> }
                                   |     is connected to { <outer exit point> }
                                              is connected to { <inner exit points> } }
```

The <solid association symbol> is attached at one end to the <frame symbol> of the enclosing diagram and the <outer entry points> or <outer exit point> is placed nearby outside this <frame symbol> of the enclosing diagram. The <solid association symbol> is attached at the other end to a <state partition area> and the <inner entry point> or <inner exit points> is placed nearby outside the <state symbol> or <dashed state symbol> for the <state partition area>. The <outer entry points> shall refer only to names defined as state entry points of the enclosing diagram. An <outer exit point> shall refer only to a name defined as a state exit point of the enclosing diagram. An <inner entry point> shall refer only to a name defined as a state entry point of the <state partition area>. The <inner exit points> shall refer only to names defined as a state exit point of the <state partition area>.

```
<outer entry points> ::=
                         <state entry points> | default

<inner entry point> ::=
                              <state entry point> |  ( <state entry point> )
                         |    default
                         |    with <nextstate parameters>

<outer exit point> ::=
                              <state exit point> |  ( <state entry point> ) | default

<inner exit points> ::=
                         <state exit points> | default
```

Each <state partition connection area> represents one or more *Connection-definition* items of the *State-partition* for the <state partition area> that is attached to the <solid association symbol> of the <state partition connection area>.

For each <state entry point> of the <state entry points> of <outer entry points> of the <state partition connection area>, there is an *Entry-connection-definition* where the *Outer-entry-point* is the *Name* for the *State-entry-point-definition* of the <state entry point>. If the <outer entry points> is **default**, the *Outer-entry-point* is the unique anonymous *Name* for the unlabelled state entry. The *Nextstate-parameters* of the *Inner-entry-point* for each of these *Entry-connection-definition* items is represented by the <inner entry point>. If the <inner entry point> is a <state entry point> or bracketed <state entry point> this represents the *State-entry-point-name* of the *Nextstate-parameters*. If the <inner entry point> is the keyword **default**, the *Actual-parameters* of the *Nextstate-parameters* are empty and the *State-entry-point-name* of the *Nextstate-parameters* is the unique anonymous *Name* for the unlabelled state entry. Otherwise <nextstate parameters> represents the *Nextstate-parameters*.

If a *State-entry-point-definition* of the *Composite-state-type-definition* directly enclosing the *State-aggregation-node* is not named in the *Outer-entry-point* of at least one *Entry-connection-definition* of a *State-partition*, there is an implicit *Entry-connection-definition*. In this *Entry-connection-definition*, the *Outer-entry-point* is the *Name* for the otherwise unconnected *State-entry-point-definition* of the enclosing *Composite-state-type-definition*, and the *Actual-parameters* of the *Nextstate-parameters* are empty and the *State-entry-point-name* of the *Nextstate-parameters* is the unique anonymous *Name* for the unlabelled state entry.

For each <state exit point> of the <state exit points> of <inner exit points> of the <state partition connection area>, there is an *Exit-connection-definition* where the *Inner-exit-point* is the *Name* for the *State-exit-point-definition* of the <state exit point>. If the <inner exit points> is **default**, the *Inner-exit-point* is the unique anonymous *Name* for the unlabelled state exit. The *Outer-exit-point* for each of these *Exit-connection-definition* items is represented by the <outer exit point>. If the <outer exit point> is a <state exit point> or bracketed <state exit point> this represents the *State-exit-point-name* of the *Outer-exit-point*. If the <inner entry point> is the keyword **default** the *State-entry-point-name* of the *Inner-entry-point* is the unique anonymous *Name* for the unlabelled state entry.

If the *State-exit-point-definition* of the composite state identified by the *Composite-state-type-identifier* of a *State-partition* is not otherwise named in an *Exit-connection-definition* for that *State-partition*, there is an implicit *Exit-connection-definition*. In this *Exit-connection-definition*, the *State-exit-point-name* of the *Inner-exit-point* is the *Name* for the otherwise unconnected *State-exit-point-definition*, and the *Outer-exit-point* is the unique anonymous *Name* for the unlabelled state exit.

Semantics

If a *Composite-state-type-definition* contains a *State-aggregation-node,* the composite states of each *State-partition* are interpreted in an interleaving manner at the transition level. Each transition runs to completion before another transition is interpreted. The creation of a composite state with state partition implies the creation of each contained *State-partition* and its connections. If the *Composite-state-type-definition* of a *State-partition* has *Composite-state-formal-parameter*s, these formal parameters are undefined when the state is entered, except in the case there is an *Inner-entry-point* of an *Entry-connection-definition* for the *State-partition* with non-empty *Actual-parameters* in its *Nextstate-parameters*.

In the absence of named entry points or explicit *Entry-connection-definition* items, each unlabelled *State-start-node* of each of the partitions is interpreted in any order as the default entry point of the composite state when the state aggregation is entered through a *Nextstate-node* has no *State-entry-point-name*. Otherwise, for the more general case for each partition either *State-start-node* or a *Named-start-node* (an additional entry point) of the partition composite state is interpreted.

When a state aggregation is entered the *Nextstate-parameters* are interpreted in the same way as entering any other composite state: the variables for the formal parameters have values assigned or are left undefined depending on the actual parameters present. The entry procedure (if any) for the state aggregation is then interpreted.

If the state aggregation is entered through the *Outer-entry-point* of an *Entry-connection-definition*, for each state partition the *Nextstate-parameters* of the corresponding *Inner-entry-point* is interpreted. If the actual parameters of the *Nextstate-parameters* are empty, the formal parameters are undefined when the state is entered. Otherwise the actual parameters (which could refer to formal parameter variables of the state aggregation) are evaluated and assigned to the corresponding formal parameter variables of the state partition. If the *State-entry-point-name* of the *Nextstate-parameters* is absent, or if it is the unique anonymous *Name* for *State-start-node*, the state partition is entered via the *State-start-node*. Otherwise the state partition is entered via the *Named-start-node* given by the *State-entry-point-name*. Each of the state partitions are entered in an undetermined order, after the entry procedure of the state aggregation is completed.

If there are signals in the complete valid input set of the *Composite-state-type-definition* where a *State-aggregation-node* occurs that are not consumed by any *State-partition* of a *State-aggregation-node*, there is an implied additional *State-partition*. The implied *Composite-state-type-definition* for this *State-partition* has a *Composite-state-graph* that is a *State-transition-graph* with a single unlabelled *State-start-node* with a transition to a single *State-node*. This *State-node* has an *Input-node* for each signal handled (including those for exported procedures and exported variables) and a *Continuous-signal*. The *Transition* of each *Input-node* is an empty transition back to the state. The *Continuous-signal* has a *Continuous-expression* that is a logical 'and' that accesses implicit Boolean variables, one for each (non-implicit) partition. These variables are shared by all partitions and each initialized to False. When each of the other partitions has interpreted either an *Action-return-node* or *Named-return-node*, its implicit Boolean variable is set to True. The *Transition* of the *Continuous-signal* contains just an *Action-return-node*. Therefore, when other partitions have completed, and the additional partition has consumed all the signals for it (if any) in the input port, the partition exits through an *Action-return-node*.

When each and every partition has interpreted (in any order) an *Action-return-node* or *Named-return-node*, the partitions exit the composite state. The *Exit-connection-definition* set associates the exit points from the partitions with the exit points of the state aggregation. If different partitions exit the composite state through different exit points, the exit point of the composite state is chosen in a non-deterministic way. The exit procedure of the state aggregation is interpreted after all state partitions have been completed. Signals in the input signal set of a partition that completed its return node are saved while all other partitions are completed and therefore are still in the input port when a transition is taken from the aggregation.

The nodes of the state partition graphs are interpreted in the same manner as the equivalent nodes of an agent, or procedure graph, with the only difference that they have disjoint input signal sets. The state partitions share the same input port as the enclosing agent.

An input transition associated with a composite state application containing a *State-aggregation-node* applies to all states of all state partitions, and it implies a default termination of all these. If such a transition terminates with a *Nextstate-node* with **HISTORY**, all partitions re-enter into their respective substates.

11.11.3 State connection point

State connection points (state entry points and state exit points) are defined for composite states and the state connection points represent connection points for entry and exit of a composite state.

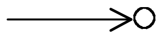
Abstract grammar

State-entry-point-definition	::	*Name*
State-exit-point-definition	::	*Name*

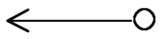
Concrete grammar

```
<state connection point area> ::=
                  <state connection point symbol 1>
                  is associated with { <state entry points> }
            |     <state connection point symbol 2>
                  is associated with { <state exit points> }
```

<state connection point symbol 1> ::=



<state connection point symbol 2> ::=



<state entry points> ::=
        <state entry point>
    | **(** <state entry point> { **,** <state entry point> }* **)**

<state exit points> ::=
        <state exit point>
    | **(** <state exit point> { **,** <state exit point> }* **)**

<state entry point> ::=
        <<u>state entry point</u> name>

A <state entry point> represents a *State-entry-point-definition*.

<state exit point> ::=
        <<u>state exit point</u> name>

A <state exit point> represents a *State-exit-point-definition*.

In <state connection point symbol 1> and <state connection point symbol 2>, the centre of the circle is placed on the edge of the <frame symbol> to which it is connected.

NOTE – The reason for the brackets around the state entry or exit points is to make it easy to see the beginning and end of the list. In the case of a single point it is allowed to omit the brackets, because there has to be at least one point and this should be easy to find.

Semantics

A *State-entry-point-definition* defines a state entry point of a *Composite-state-type-definition*. Each *Composite-state-type-definition* has a *State-entry-point-definition* with a unique anonymous *Name* for an unlabelled state entry, which is the unlabelled *State-start-node* of a *Composite-state-type-definition* with a *Composite-state-graph*.

A *State-exit-point-definition* defines a state exit point of a *Composite-state-type-definition*. Each *Composite-state-type-definition* has a *State-exit-point-definition* with a unique anonymous *Name* for an unlabelled state exit, which is the unlabelled *Return-node* of the *Composite-state-type-definition* with a *Composite-state-graph*.

11.11.4 Connect

A connect is allowed to have a state exit name, and the transition is allowed to be virtual.

Abstract grammar

Connect-node	::	[*State-exit-point-name*] *Transition*
State-exit-point-name	=	*Name*

Connect-node is extended compared with Basic SDL-2010 to include an optional *State-exit-point-name* for the corresponding connect for a labelled return. The *State-exit-point-name* shall be the *Name* of a *State-exit-point-definition* for the *Composite-state-type-definition* identified by the *State-node* that contains the *Connect-node*.

Each *Connect-node* in the *Connect-node-**set*** of a composite state application shall either be the only *Connect-node* without a *State-exit-point-name* or have a *State-exit-point-name* that is different from every other *Connect-node* in the *Connect-node-**set***.

Concrete grammar

```
<connect association area> ::=
                    <solid association symbol> is associated with { [<virtuality>] [<connect list>] }
                    is followed by <exit transition area>
```

<connect association area> is extended compared with Basic SDL-2010 to optionally include <virtuality> and optionally include a <connect list>.

```
<connect list> ::=
                    <state exit point list>

<state exit point list> ::=
                    <state exit point>
```

If a <connect list> is given, a <state exit point> represents the *State-exit-point-name* of the *Connect-node*.

Semantics

A *Connect-node* without a *State-exit-point-name* corresponds to an unlabelled *Return-node* in a composite state: that is, the unique anonymous *Name* for the unlabelled *Return-node* of the *Composite-state-type-definition*.

A *Connect-node* with a *State-exit-point-name* corresponds to a labelled *Return-node* in a composite state.

11.12 Transition

11.12.1 Transition body

Abstract grammar

```
Graph-node                    ::    { Task-node
                                     | Output-node
                                     | Create-request-node
                                     | Call-node
                                     | Compound-node
                                     | Set-node
                                     | Reset-node }
```

Graph-node is extended compared with Basic SDL-2010 to include *Compound-node*, which allows a <task area> to contain a number of textual statements.

```
Terminator                    ::    { Nextstate-node
                                     | Stop-node
                                     | Return-node
                                     | Join-node
                                     | Continue-node
                                     | Break-node }
```

Terminator is extended compared with Basic SDL-2010 to include *Continue-node* and *Break-node*, which are used in *Compound-node*.

Concrete grammar

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

<terminator area> is extended compared with Basic SDL-2010 to include <transition option area> for generic descriptions.

```
<action area> ::=
                    <task area>
                |   <output area>
                |   <create request area>
                |   <procedure call area>
                |   <remote procedure call area>
```

<action area> is extended compared with Basic SDL-2010 to include <remote procedure call area> for generic descriptions.

11.12.2 Transition terminator

11.12.2.1 Nextstate

Abstract grammar

Nextstate-parameters	::	*Actual-parameters*
		[*State-entry-point-name*]

Nextstate-parameters is extended compared with Basic SDL-2010 to allow an optional *State-entry-point-name* for entering a composite state via a named start. The *State-entry-point-name* (if given) shall be defined for the *Composite-state-type-definition* of the state.

Concrete grammar

```
<nextstate parameters> ::=
                [<actual parameters>] [ via <state entry point name> ]
```

<nextstate parameters> is extended compared with Basic SDL-2010 to allow an optional <state entry point name> that represents the *State-entry-point-name* of the *Nextstate-parameters*.

Semantics

If a *State-entry-point-name* is given, the next state is a composite state, and interpretation continues with the *State-start-node* that has the same name in the *Composite-state-graph*.

When a *Dash-nextstate* with **HISTORY** is interpreted, if interpretation re-enters a composite state, its entry procedure is invoked.

11.12.2.2 Join

Join is as defined in [ITU-T Z.101].

11.12.2.3 Stop

Semantics

A *Stop-node* in a *Compound-node* propagates outwards through the invocation of the *Compound-node* until the containing agent is reached.

11.12.2.4 Return

Abstract grammar

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

Return-node is extended compared with Basic SDL-2010 to allow a *Named-return-node* as an alternative.

Named-return-node	::	*State-exit-point-name*

A *Named-return-node* shall be directly contained in a *Composite-state-graph* that has a *State-exit-point-definition* with the same *Name* as the *State-exit-point-name*.

Concrete grammar

```
<return area> ::=
                        <return symbol>
                        is associated with { <return body> | via <state exit point> }
```

<return area> is extended compared with Basic SDL-2010 to allow **via** <state exit point> as an alternative that represents the *State-exit-point-name* of a *Named-return-node*.

Semantics

When a *Named-return-node* is interpreted, local variables cease to exist and control is transferred to the context of the state composition.

In a composite state that is not a partition of a state aggregation, after interpretation of the *Named-return-node* interpretation continues at the *Connect-node* with the same name.

In a composite state that is a partition of a state aggregation, after interpretation of the *Named-return-node* control is transferred to the *State-aggregation-node*. The *Exit-connection-definition* for the *State-partition* with the *State-exit-point-name* as *Inner-exit-point* is used to determine the intended *Outer-exit-point*. If the state aggregation is not contained in another state aggregation, a *Connect-node* with the same name as the *Outer-exit-point* is identified. When all the partitions of the *State-aggregation-node* have terminated, one identified *Outer-exit-point* is taken. If more than one *Outer-exit-point* has been identified by the partitions, the basis of the choice is not defined by SDL-2010.

11.13 Action

11.13.1 Task

Concrete grammar

The syntax for <non terminating statements> of <task body> of <task area> of Basic SDL-2010 is extended to allow more than one statement in <non terminating statements> and for statements other than <assignment>, <set statement> and <reset statement>. Each <non terminating statement> of <task body> of <task area> represents an element of the *Graph-node* list for the *Transition* of the <transition area> containing the <task symbol> in the order that each <non terminating statement> occurs in the <task area>.

Semantics

The interpretation of a *Compound-node* is given in clause 11.14.1.

11.13.2 Create

Create is as defined in [ITU-T Z.101].

11.13.3 Procedure call

Abstract grammar

Call-node	::	[**THIS**] *Procedure-identifier* *Actual-parameters*
Value-returning-call-node	::	[**THIS**] *Procedure-identifier* *Actual-parameters*

Basic SDL-2010 is extended to include the optional **THIS** to indicate in a specialized procedure that the specialized procedure should be call rather than the unspecialized procedure.

Concrete grammar

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

Basic SDL-2010 is extended to include **this** which represents **THIS**.

If the <procedure type expression> has actual context parameters, there is an implicitly created procedure definition with an anonymous name in the enclosing agent and the *Procedure-identifier* identifies this procedure. If the <procedure type expression> does not have actual context parameters, <procedure type expression> is limited to <base type>, which is a <procedure identifier>.

Semantics

If **THIS** is present and the procedure is specialized, the *Procedure-identifier* refers to the identifier of the specialized procedure. For a procedure that is not specialized or if **THIS** is absent, the *Procedure-identifier* refers to the identifier of the procedure that is not specialized.

11.13.4 Output

Concrete grammar

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

<communication constraints> is extended compared with Basic SDL-2010 to include a <timer communication constraint>. The <communication constraints> in an <output body> shall not contain a <timer communication constraint>. A <communication constraints> in <remote procedure call body> shall contain no more than one <timer communication constraint> (see clause 10.5). A <communication constraints> in an <import expression> shall contain no more than one <timer communication constraint> (see clause 10.6).

11.13.5 Decision

Abstract grammar

```
Decision-node                =   Decision-body
                             |   Any-decision
```

Basic SDL-2010 is extended to allow *Any-decision* in a *Decision-node*.

```
Any-decision                 ::  Transition-set
```

Concrete grammar

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

The <decision area> of Basic SDL-2010 is extended to include **any** for a non-deterministic decision.

If the <decision symbol> of a <decision area> contains the keyword **any**, the <decision area> represents an *Any-decision*.

Semantics

An *Any-decision* interpretation transfers the interpretation to a *Transition* of the *Any-decision* selected on an arbitrary basis. Each interpretation of an *Any-decision* selects a *Transition* on a random basis, but the distribution of the selections is not further defined, though it is expected that normally after a large number of interpretations each *Transition* would have been selected at least once.

11.14 Statement lists

The purpose of statement lists is to allow concise textual descriptions to be combined with the graphical representation for actions and local variable definitions. The statements are effectively interpreted in the order they occur (left to right, top to bottom) within the text, for example, in a <task area>.

Concrete grammar

```
<statements> ::=
                 <non terminating statements> [ <end>+ <terminating statement> ]
            |    <terminating statement>
```

Basic SDL-2010 is extended to include <statements>, which is a list of statements used in a compound statement.

```
<non terminating statements> ::=
                 <non terminating statement>
                     { <end>+ <non terminating statement> }*
```

The <non terminating statements> list of Basic SDL-2010 is extended to allow more than one statement.

```
<non terminating statement> ::=
                 <statement>
            |    <compound statement>
            |    <loop statement>
            |    <decision statement>
```

The <non terminating statement> of Basic SDL-2010 is extended to include <compound statement>, <loop statement> and <decision statement>.

```
<statement> ::=
                 <assignment statement>
            |    <set statement>
            |    <reset statement>
            |    <output statement>
            |    <create statement>
            |    <export statement>
            |    <call statement>
```

The <statement> of Basic SDL-2010 is extended to include <output statement>, <create statement>, <export statement> and <call statement>.

```
<terminating statement> ::=
                 <return statement>
            |    <stop statement>
            |    <break statement>
```

NOTE 1 – <assignment statement> is defined in Basic SDL-2010 and represents an *Assignment*.

NOTE 2 – <set statement> is defined in Basic SDL-2010 and represents a *Set-node*.

NOTE 3 – <reset statement> is defined in Basic SDL-2010 and represents a *Reset-node*.

```
<output statement> ::=
                 output <output body>
```

An <output statement> represents an *Output-node* as further discussed in clause 11.13.4.

<create statement> ::=
    create <create body>

A <create statement> represents a *Create-request-node* as further discussed in clause 11.13.2.

<export statement> ::=
    export <export body>

NOTE 4 – The *Model* for <export statement> is given in clause 10.6.

<call statement> ::=
    [**call**] { <procedure call body> | <remote procedure call body> }

A <call statement> represents a *Call-node* as further discussed in clause 11.13.3.

The keyword **call** shall not be omitted if the <call statement> is syntactically ambiguous with an operation application or variable with the same name.

NOTE 5 – This ambiguity is not resolved by context.

<return statement> ::=
    return <return body>

A <return statement> represents a *Return-node* as further discussed in clause 11.12.2.4.

A <return statement> is only allowed within a procedure or within an operation, which in Comprehensive SDL-2010 means within an <procedure diagram> or an <operation diagram>.

NOTE 6 – Textual alternatives to <procedure diagram> and <operation diagram> are defined in [ITU-T Z.103].

<stop statement> ::=
    stop

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

11.14.1 Compound and loop statements

A compound node encapsulates a number of other nodes so that they are treated as a single node with local variables and allows the encapsulated nodes to be interpreted iteratively. A compound node has a connector name, used in continue statements and break statements to go back to the start or re-enter the encapsulated nodes.

A <compound statement> groups multiple statements into a single statement, which optionally defines local variables.

A <loop statement> provides iteration of a <loop body statement>, with an arbitrary number of loop variables, which optionally are locally defined. The loop variables are stepped as specified by the <loop step>. The loop variables are both to generate successive results and to accumulate results. When the <loop statement> terminates, a <finalization statement> is optionally interpreted in the context of any locally defined loop variables.

The <loop body statement> is part of a <loop statement> and is interpreted repeatedly, controlled by the <loop clause>, if any. A <loop variable indication> in a <loop clause> provides declaration and initialization of local loop variables. The scope and lifetime of any variable defined in a <loop variable indication> is that of the <loop statement>. A <loop variable definition> in a <loop variable indication> defines a loop variable and if initialization is present as an <expression> in the <loop variable definition>, the expression is evaluated only once before the first interpretation of the loop body. An alternative to <loop variable definition> in a <loop variable indication> is a visible variable identified as a loop variable by a <variable identifier>, optionally with an expression assigned to it. Before each iteration, all <Boolean expression> elements of the <loop clause> are evaluated. Interpretation of the <loop statement> is terminated if any one <Boolean expression> element returns false. If there is no <Boolean expression> present, interpretation of the

<loop statement> continues until the <loop statement> exits to a non-local connector. If a <loop variable indication> is present in that <loop clause>, at the end of each iteration the <loop step> in that <loop clause> is interpreted and the result assigned to the loop variable. If a <loop variable indication> was not present in a <loop clause>, or if a <loop step> was not present, no assignment statement to a loop variable is performed. The <loop variable indication>, <Boolean expression>, and <loop step> are optional. Assignments to a loop variable in the <loop body statement> are not allowed.

Interpretation of the loop body terminates when a **break** statement without a connector is reached. Reaching a **continue** statement without a connector causes interpretation of the loop to jump immediately to the next iteration.

If a <loop statement> is terminated "normally" (that is, by a <Boolean expression> evaluating to the predefined Boolean value false), the <finalization statement> is interpreted. A loop variable is visible and retains its result when the <finalization statement> is interpreted. A break or continue statement without a connector within the <finalization statement> terminates the next outer <loop statement>.

A <loop continue statement> causes interpretation to be transferred to a point after initializations within the compound node with the matching connector name.

A <break statement> causes interpretation to be transferred to the point following the compound node with the matching connector name.

Abstract grammar

Compound-node	::	*Connector-name* *Variable-definition-**set*** *Init-graph-node** *While-graph-node* *Transition* *Step-graph-node**

No element of the *Variable-definition-**set*** of the *Compound-node* has a *Constant-expression*.

Init-graph-node	=	*Graph-node*
While-graph-node	=	*Expression** [*Finalization-node*]
Finalization-node	=	*Graph-node*

Each *Expression* in a *While-graph-node* shall be a Boolean *Expression*.

Step-graph-node	=	*Graph-node*

A *Graph-node* of a *Step-graph-node* shall be a *Task-node*.

Continue-node	::	*Connector-name*

A *Continue-node* shall be contained in a *Compound-node* that has been labelled with the given *Connector-name*.

Break-node	::	*Connector-name*

A *Break-node* shall be contained in a *Compound-node* that has been labelled with the given *Connector-name*.

Every *Terminator* of *Transition* in a *Compound-node,* regardless of whether it is the *Terminator* of the *Transition* of the *Compound-node* or the *Terminator* for a *Decision-answer* or *Else-answer* within the *Compound-node,* shall be a *Return-node* or *Continue-node* or *Break-node*.

Concrete grammar

<compound statement> ::=
    [<connector name> :] [<comment body>]
        <left curly bracket>
            [<variable definitions> <end>]
            [<statements>] <end>*
        <right curly bracket>

The <compound statement> represents a *Compound-node*. If the <compound statement> has a <connector name> this represents the *Connector-name*; otherwise a newly created anonymous name represents the *Connector-name*.

If the <statements> list is omitted, the *Transition* is an empty *Graph-node* list followed by a *Break-node* with the *Connector-name* as the *Terminator* of the *Transition*.

If the <statements> list does not end in a <terminating statement>, the *Transition* is a *Graph-node* list represented by the <non terminating statements> of the <statements> list followed by a *Break-node* with the *Connector-name* as the *Terminator* of the *Transition*. If the <statements> list ends in a <terminating statement>, the *Transition* is a *Graph-node* list represented by the <non terminating statements> of the <statements> list followed by the *Terminator* represented by the <terminating statement>.

For a <compound statement> the *Expression* list of the *While-graph-node* is empty, the *Finalization-node* of the *While-graph-node* is omitted, and the *Step-graph-node* list is empty.

A <compound statement> creates its own scope for any variables defined in the statement and the connector name.

<variable definitions> ::=
    <variable definition statement> { <end> <variable definition statement> }*

<variable definition statement> ::=
    dcl <local variables of sort> { **,** <local variables of sort> }*

<local variables of sort> ::=
    <aggregation kind> <variable name> { **,** <variable name>}* <sort>
        [<is assigned sign> <expression>]

Each <variable name> and subsequent <sort> of <local variables of sort> of a <variable definition statement> represents the *Variable-name*, *Sort-reference-identifier* and *Aggregation-kind* of an element of the *Variable-definition-**set*** of the *Compound-node*, as described for *Variable-definition* in clause 12.3.1 of [ITU-T Z.101]. Each <variable name> in <local variables of sort> with an <is assigned sign> and <expression> represents a *Task-node* that contains an *Assignment* where the <variable name> represents the *Variable-identifier* and the <expression> represents the *Expression*. The *Init-graph-node* list is each *Task-node* from the <variable definitions> of the <compound statement> in the order they occur from left to right.

<loop statement> ::=
    [<connector name> :]
    loop (<loop clause> { **;** <loop clause> }* **)**
        <loop body statement> [**then** <finalization statement>]

A <loop statement> represents a *Compound-node*. If the <loop statement> has a <connector name> this represents the *Connector-name*; otherwise a newly created anonymous name represents the *Connector-name*.

A <loop statement> creates its own scope for variables defined in the statement and the connector name.

<loop clause> ::=
    [<loop variable indication>]
    [**,** [<Boolean expression>]
        <loop step>]

Each <Boolean expression> in a <loop clause> represents a Boolean *Expression* in the *While-graph-node* of the *Compound-node* for the <loop statement> in the order of each <loop clause> in the <loop statement>.

```
<loop variable indication> ::=
                    <loop variable definition>
                |   <loop variable indication identifier>
```

```
<loop variable indication identifier> ::=
                    <variable identifier> [ <is assigned sign> <expression> ]
```

A <variable identifier> of a <loop variable indication> shall not identify a variable defined in a <loop variable definition>.

Each <variable identifier> and associated <expression> in a <loop variable indication> represents the *Variable-identifier* and *Expression* of an *Assignment* in a *Task-node* of the *Init-graph-node* list of the *Compound-node*. The <variable identifier> in a <loop variable indication> also represents the *Variable-identifier* in the *Step-graph-node* for the <loop step> following the <loop variable indication>. The <variable identifier> in the <loop variable indication> also represents the *Variable-identifier* in any *Step-graph-node* for the <loop step> following the <loop variable indication>.

```
<loop variable definition> ::=
                    dcl <aggregation kind> <variable name> <sort> <is assigned sign> <expression>
```

Each <aggregation kind> <variable name> <sort> set in a <loop variable definition> represents the *Aggregation-kind*, *Variable-name* and *Sort-reference-identifier* of an element of the *Variable-definition-**set*** of the *Compound-node*. Each <variable name> and associated <expression> in a <loop variable definition> also represents the *Variable-identifier* and *Expression* of an *Assignment* in a *Task-node* of the *Init-graph-node* list of the *Compound-node*. The <variable name> in a <loop variable definition> also represents the *Variable-identifier* in any *Step-graph-node* for the <loop step> following the <loop variable indication>.

The *Task-node* items from the <loop variable indication> items occur in the *Init-graph-node* list in the same order (left to right, top to bottom) as the <loop clause> items in the <loop statement>.

```
<loop step> ::=
                    [ , [<expression>] ]
```

Each <expression> of a <loop step> represents an *Expression* of an *Assignment* in a *Task-node* of the *Step-graph-node* list of the *Compound-node*. The *Variable-identifier* of the *Assignment* is the one represented by the <variable identifier> or <variable name> of the preceding <loop variable indication>; therefore if a <loop step> of a <loop clause> has an <expression> the <loop variable indication> of the <loop clause> shall not be omitted.

The *Task-node* items from the <loop step> items occur in the *Step-graph-node* list in the same order (left to right, top to bottom) as the <loop clause> items in the <loop statement>.

```
<loop body statement> ::=
                    <statement in loop>
```

A <loop body statement> represents the *Transition* of the *Compound-node*. The *Transition* is the *Graph-node* represented by the <statement in loop> (a <non terminating statement> or <loop statement> or <loop compound statement> or <loop decision statement>) followed by a *Continue-node* with the *Connector-name* of the *Compound-node*.

There shall be no assignments to a loop variable in the <loop body statement>.

NOTE – If a variable defined outside the loop is used as a loop variable and a procedure is called within a loop statement, it is possible that the procedure call changes the loop variable and this is probably a design error. Therefore it is suggested to avoid such constructs, and any tool implementing the language should provide a warning if this situation is detected.

```
<statement in loop> ::=
                                <statement>
                        |       <loop statement>
                        |       <loop compound statement>
                        |       <loop decision statement>

<loop compound statement> ::=
                                 [ <connector name> : ] [ <comment body> ]
                                <left curly bracket>
                                        [ <variable definitions> <end> ]
                                        [ <loop statements> ] <end>*
                                <right curly bracket>
```

A <loop compound statement> differs from a <compound statement> because it uses <loop statements> instead of <statements>. A <loop compound statement> represents a *Compound-node* in the same way as a <compound statement> except that the *Transition* represented by <loop statements> optionally ends with a *Break-node* for a <loop break statement> or a *Continue-node* for a <loop continue statement>.

A <loop compound statement> creates its own scope for variables defined in the statement and the connector name.

```
<loop statements> ::=
                                <statement in loop> { <end>+ <statement in loop> }*
                                        [ <end>* <loop terminating statement> ]
                        |       <loop terminating statement>
```

A <loop statements> list differs from a <statements> because it optionally ends with a <loop terminating statement> instead of <terminating statement>, and also because it has <statement in loop> instead of <statement>. The <loop terminating statement> allows <loop break statement> and <loop continue statement> as well as <terminating statement>.

The <loop statements> list of a <loop compound statement> represents the *Transition* of the *Compound-node* in the same way as the <statements> of a <compound statement>, except that a *Break-node* for a <loop break statement> or a *Continue-node* for a <loop continue statement> is also allowed as the *Terminator* of the *Transition*.

```
<loop terminating statement> ::=
                                <terminating statement>
                        |       <loop break statement>
                        |       <loop continue statement>

<loop break statement> ::=
                                break
```

A <loop break statement> represents a *Break-node* with the *Connector-name* of the immediately enclosing *Compound-node* for a <loop statement>.

```
<loop continue statement> ::=
                                continue [ <connector name>  ]
```

A <loop continue statement> without a <connector name> represents a *Continue-node* with the *Connector-name* of the immediately enclosing *Compound-node* for a <loop statement>. A <connector name> in a <loop continue statement> represents the *Connector-name* of the *Continue-node*.

```
<finalization statement> ::=
                                <statement>
                        |       <compound statement>
```

A <finalization statement> represents the *Finalization-node* of the *While-graph-node* of the *Compound-node* of the <loop statement>. The <statement> or <compound statement> of the <finalization statement> represents the *Graph-node* of the *Finalization-node*.

```
<break statement> ::=
                              break <connector name>
```

A <break statement> represents a *Break-node. A* <connector name> in a <break statement> represents the *Connector-name* of the *Break-node*.

Semantics

The interpretation of a *Compound-node* proceeds as follows.

a) A local variable is created for each *Variable-definition* in the *Variable-definition-set.*

b) The ***Init-graph-node*** list is interpreted.

c) If the *Compound-node* does not have a *While-graph-node*, interpretation continues at step (d), otherwise each *Expression* of the *While-graph-node* is interpreted in the order of the *Expression* list until

   i) either one *Expression* interprets as the Boolean false value and the *Finalization-node* (if there is one) is interpreted, followed by termination of the *Compound-node* (as in step (f) item (iv) below) and interpretation continuing at the node following the *Compound-node*;

   ii) or every *Expression* has been interpreted and further interpretation continues at step (d).

d) The *Transition* is interpreted, which either results in interpretation of a *Terminator* that is a *Continue-node* as in step (e), or the interpretation of a *Terminator* that is a *Break-node, Continue-node* or *Return-nod*e as in step (f).

e) When a *Continue-node* with a *Connector-name* matching the *Connector-name* of the *Compound-node* is interpreted, the *Step-graph-node* list is interpreted and further interpretation continues at step (c).

f) When the interpretation of the *Compound-node* terminates, all variables created by the interpretation of the *Compound-node* cease to exist. Interpretation of a *Compound-node* terminates:

   i) when a *Break-node* is interpreted; or

   ii) when a *Continue-node* with a *Connector-name* different from the *Connector-name* of the *Compound-node* is interpreted; or

   iii) when a *Return-node* is interpreted; or

   iv) when an *Expression* of the *While-graph-node* interprets as the `Boolean` value `false`.

g) Hereafter, interpretation continues as follows:

   i) If the interpretation of the *Compound-node* terminated due to the interpretation of a *Break-node* with a *Connector-name* matching the *Connector-name* of the *Compound-node*, interpretation continues at the node following the *Compound-node*; otherwise

   ii) If the interpretation of the *Compound-node* terminated due to the interpretation of a *Break-node* (that does not have a *Connector-name* matching the *Connector-name* of the *Compound-node*), *Continue-node* or *Return-node*, then the interpretation continues with interpretation of the *Break-node, Continue-node* or *Return-node*, respectively, at the point of invocation of the *Compound-node*; otherwise

   iii) If the interpretation of the *Compound-node* terminated because an *Expression* of the *While-graph-node* interprets as the `Boolean` value `false`, interpretation continues at the node following the *Compound-node*.

A *Continue-node* causes the interpretation to be transferred to *While-graph-node* or (if there is no *While-graph-node*) *Transition* of the *Compound-node* with the matching *Connector-name* as further described above for the interpretation of a *Compound-node*.

A *Break-node* causes the interpretation to be transferred to the *Graph-node* following the *Compound-node* with the matching *Connector-name* as further described above for the interpretation of a *Compound-node*.

11.14.2 Decision statement

The decision statement is a concise form of decision. An expression for the decision is evaluated and the answer whose range condition contains the result of the expression is interpreted. Overlapping range conditions are not allowed. If there is no match and an alternative statement exists, the alternative statement is interpreted. If there is no match and an alternative statement does not exist, interpretation continues after the decision statement.

Concrete grammar

```
<decision statement> ::=
                [ <connector name> : ] decision ( <question> ) [ <comment body> ]
                <left curly bracket>
                        <decision statement body>
                <right curly bracket>
```

A <decision statement> represents a *Compound-node*. If the <decision statement> has a <connector name> this represents the *Connector-name* of the *Compound-node*; otherwise a newly created anonymous name represents the *Connector-name*. The *Variable-definition-**set***, *Init-graph-node* list, *Expression* list of the *While-graph-node*, and *Step-graph-node* list of the *Compound-node* are empty. There is no *Finalization-node* in the *While-graph-node*. The *Transition* is an empty *Graph-node* list followed by a *Decision-node*.

A <decision statement> creates its own scope for variables defined in the statement and the connector name.

```
<decision statement body> ::=
                <algorithm answer part>+ [<algorithm else part>]

<algorithm answer part> ::=
                ( <answer> ) <colon>
                [ <non terminating statement> | <terminating statement> ]
```

Each <algorithm answer part> of a <decision statement> represents a *Decision-answer* of the *Decision-node* of the *Transition* of the *Compound-node*. If the <algorithm answer part> is a <non terminating statement>, the *Transition* of the *Decision-answer* is the *Graph-node* represented by the <non terminating statement> followed by a *Break-node* with the *Connector-name* of the *Compound-node*. If the <algorithm answer part> is a <terminating statement>, the *Transition* of the *Decision-answer* is the *Terminator* represented by the <terminating statement>.

If there is no <non terminating statement> or <terminating statement> in the <algorithm answer part>, the *Transition* of the *Decision-answer* is an empty *Graph-node* list followed by a *Break-node* with the *Connector-name* of the *Compound-node*.

```
<algorithm else part> ::=
                else <colon> [ <alternative statement> ]
```

An <algorithm else part> of a <decision statement> represents the *Else-answer* of the *Decision-node* of the *Transition* of the *Compound-node*. If the <alternative statement> is a <non terminating statement>, the *Transition* of the *Else-answer* is the *Graph-node* represented by the <non terminating statement> followed by a *Break-node* with the *Connector-name* of the *Compound-node*. If the <alternative statement> is a <terminating statement>, the *Transition* of the *Else-answer* is the *Terminator* represented by the <terminating statement>.

If there is no <alternative statement> or no <algorithm else part> in the <decision statement> the *Transition* of the *Else-answer* contains only a *Break-node* with the *Connector-name* of the *Compound-node*.

```
<alternative statement> ::=
                    { <non terminating statement> | <terminating statement> }

<loop decision statement> ::=
                    [ <connector name> : ] decision ( <question> ) [ <comment body> ]
                    <left curly bracket>
                         <loop decision statement body>
                    <right curly bracket>
```

A <loop decision statement> differs from a <decision statement> only because it uses <loop decision statement body> instead of <decision statement body> so that <loop break statement> and <loop continue statement> are allowed in the loop. Similarly <loop answer part>, <loop else part>, <loop alternative statement>, <statement in loop> and <loop terminating statement> are used instead of <algorithm answer part>, <algorithm else part>, <alternative statement>, <non terminating statement> and <loop terminating statement>, respectively. A <loop decision statement> represents a *Compound-node* in the same way as <decision statement> except that a *Transition* of a *Decision-answer* or *Else-answer* is a *Break-node* for a <loop break statement> or a *Continue-node* for a <loop continue statement>.

A <loop decision statement> creates its own scope for variables defined in the statement and the connector name.

```
<loop decision statement body> ::=
                    <loop answer part>+ [<loop else part>]
```

A <loop decision statement body> differs from a <decision statement body> only because it uses <loop answer part> and <loop else part> instead of <algorithm answer part> and <algorithm answer part>, respectively, which allows <loop break statement> items and <loop continue statement> to be used in a <loop decision statement body>.

```
<loop answer part> ::=
                    ( <answer> ) <colon>
                    [ <statement in loop> | <loop terminating statement> ]

<loop else part> ::=
                    else <colon> [ <loop alternative statement> ]

<loop alternative statement> ::=
                    { <statement in loop> | <loop terminating statement> }
```

11.15 Timer

Basic SDL-2010 is extended to include a reset for all instances of a timer with parameters.

Abstract grammar

Basic SDL-2010 is extended so that number of items in the *Expression* list in a *Reset-node* is allowed to be zero (that is, the *Expression* list is empty) for a *Timer-identifier* that identifies a *Timer-definition* with a *Sort-reference-identifier* list that is not empty.

Concrete grammar

```
<reset clause> ::=
                    <timer identifier> [ ( <expression list> ) | <asterisk> ]
```

Basic SDL-2010 <reset clause> is extended to allow (as an alternative to an expression list) an <asterisk>, which represents an empty *Expression* list in the *Reset-node*.

Semantics

If the *Expression* list of a *Reset-node* is empty and timer sort list is not empty (that is, the *Sort-reference-identifier* list following the *Timer-name* in the *Timer-definition* identified by the *Timer-identifier* of the *Reset-node* is not empty), every timer of the agent with the *Timer-name* identified by the *Timer-identifier* is reset and becomes inactive.

12 Data

See Basic SDL-2010 and [ITU-T Z.104] (plus [ITU-T Z.107] for object-oriented data) if any feature defined in that Recommendation applies, except active use of data as defined below for any imperative expression that is an import expression.

12.1 Data definitions

As defined in Basic SDL-2010 and [ITU-T Z.104] (plus [ITU-T Z.107] for object-oriented data) if any feature defined in that Recommendation applies with the extension defined below for *Value-data-type-definition*.

Abstract grammar

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*]
		[*Abstract*]

Value-data-type-definition is extended to optionally allow *Abstract* to be specified. See clause 8.1.3 for more information on *Abstract*.

12.2 Use of data

As defined in Basic SDL-2010 and [ITU-T Z.104] (plus [ITU-T Z.107] for object-oriented data) if any feature defined in that Recommendation applies.

12.3 Active use of data

As defined in Basic SDL-2010 and [ITU-T Z.104] (plus [ITU-T Z.107] for object-oriented data) if any feature defined in that Recommendation applies, except for variable definition and variable access, which are extended as defined below.

12.3.1 Variable definition

Variable definition is extended for exported variables.

Concrete grammar

<variable definition> ::=
    dcl <variables of sort> {**,** <variables of sort> }* <end>
  | **dcl exported** <exported variables of sort> {**,** <exported variables of sort> }* <end>

<variable definition> is extended to allow **exported** and <exported variables of sort> for exported variables.

<exported variables of sort> ::=
    <aggregation kind> <exported variable>{ **,** <exported variable> }*
        <sort> [<is assigned sign> <constant expression>]

<exported variable> ::=
<variable name> **as** <remote variable identifier>

Two exported variables in an agent shall not mention the same <remote variable identifier>. The <remote variable identifier> of an <exported variable> is defined by a <remote variable definition>.

There is a *Variable-definition* for each <variable name> in <exported variables of sort>. 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* for the *Variable-definition* is formed in the same way as the *Constant-expression* for <variables of sort>.

12.3.2 Variable access

Variable access is extended for import expression.

Concrete grammar

```
<variable access> ::=
                         <variable identifier>
                    |    <import expression>
```

<variable access> is extended compared with Basic SDL-2010 to allow <import expression>.

Model

<import expression> is transformed to an exchange of signals before the symbol containing the <import expression> and a *Variable-access* for the implicit variable for the imported value in place of the <import expression> as described in clause 10.6. If <import expression> occurs several times in an expression, a different implicit variable is used for each occurrence.

12.3.3 Assignment

As defined in Basic SDL-2010 and [ITU-T Z.104] (plus [ITU-T Z.107] for object-oriented data) if any feature defined in that Recommendation applies.

12.3.4 Imperative expression

As defined in Basic SDL-2010 and [ITU-T Z.104] if any feature defined in that Recommendation applies.

12.3.5 Value returning procedure call

As defined in Basic SDL-2010 and [ITU-T Z.104] if any feature defined in that Recommendation applies.

13 Generic system definition

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

A generic system definition is a system definition that contains a synonym defined by an external synonym definition item (see clause 12.1.8.3 of [ITU-T Z.104]), an operation defined by an external operation definition (see clause 12.1.7 of [ITU-T Z.104]), a procedure defined by an external procedure definition (see clause 9.4 of [ITU-T Z.103]) or <informal text> in a transition option (see clause 13.2). A specific system definition is created from a generic system definition by providing results for the external synonym definition items, providing behaviour for external operation definitions and external procedure definitions, and transforming <informal text>

to formal constructs. How this is accomplished, and the relation to the abstract grammar, is not part of the SDL-2010 language definition.

13.1 Optional definition

Concrete grammar

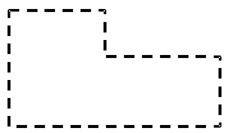
```
<select definition> ::=
                    select if ( <Boolean simple expression> ) [ <end> ]
                    {          <signal definition list>
                          |    <data definition>
                          |    <variable definition>
                          |    <timer definition>
                          |    <macro definition>
                          |    <remote variable definition>
                          |    <remote procedure definition>
                          |    <select definition>
                          |    <operation definition item> }+
                    endselect <end>

<option area> ::=
                    <option symbol> contains
                    { select if ( <Boolean simple expression> ) [ <end> ]
                              {    <agent type reference area>
                              |    <agent area>
                              |    <channel definition area>
                              |    <package text area>
                              |    <agent text area>
                              |    <procedure text area>
                              |    <composite state type reference area>
                              |    <state partition area>
                              |    <procedure reference area>
                              |    <option area> } + }

<option symbol> ::=
                    {     <dashed line symbol> is attached to <dashed line symbol>
                          <dashed line symbol> is attached to <dashed line symbol>
                          <dashed line symbol> is attached to <dashed line symbol>
                          { <dashed line symbol> is attached to <dashed line symbol> }+ }set

<dashed line symbol> ::=
                    -----------------
```

The <option symbol> shall form a dashed rectilinear polygon having solid corners, for example:



An <option symbol> logically contains the whole of any one-dimensional graphical symbol cut by its boundary (that is, with one endpoint inside).

The only visible names in a <Boolean simple expression> of a <select definition> are names of external synonyms defined outside of any <select definition>s or <option area>s and literals and operations of the data types defined within the package `Predefined` as defined in [ITU-T Z.104].

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

An <option area> is allowed to appear anywhere on a diagram, except within a body area (<composite state body area>, <procedure body area> or <operation body area>; see also agent body area in [ITU-T Z.103]). An <option area> shall contain only those areas and diagrams that are syntactically allowed at that place.

Model

Selection of optional definitions is a form of macro expansion and is done in conjunction with macro handling. Macro expansion takes place before selection is considered and therefore any contained <macro call> is expanded, before the optional definitions result is applied.

If the result of the <Boolean simple expression> is the predefined `Boolean` value `false`, the constructs contained in the <select definition> or <option symbol> are not selected. In the other case, the constructs are selected. The <select definition> and the <option area> are deleted at transformation and are replaced by the contained selected constructs, if any. Any connectors connected to an area within non-selected <option area>s are removed too. This transformation takes place before any transformation of the constructs contained in the <select definition> or <option symbol>.

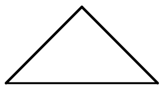
13.2 Optional transition string

Concrete grammar

```
<transition option area> ::=
                        <transition option symbol> contains <alternative question>
                        {is followed by <answer part>}+
                        is followed by <else part>

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

<transition option symbol> ::=
```



The <flow line symbol>s for **is followed by** are connected to the bottom of the <transition option symbol>.

The <flow line symbol>s originating from a <transition option symbol> are allowed to have a common originating path.

Every <constant expression> in <answer> of an <answer part> shall be a <simple expression>. Each <answer> in the <answer part> set in a <transition option area> shall be mutually exclusive with every other <answer>. If the <alternative question> is an <expression>, the *Range-condition* of each <answer> in an <answer part> shall be of the same sort as the <alternative question>.

There is a syntactic ambiguity between <informal text> and <character string> in <alternative question> and <answer>s in the <transition option area>. If the <alternative question> and all <answer>s are <character string>s, all of these are interpreted as <informal text>. If the <alternative question> or any <answer> is a <character string> and this does not match the context of the transition option, the <character string> denotes <informal text>.

Model

The <transition area> of an <answer part> is selected if the <answer> contains the result of the <alternative question>. If no <answer> contains the result of the <alternative question>, the <transition area> of the <else part> is selected.

If no <else part> is provided and none of the outgoing paths are selected, the selection is invalid.

On transformation each <transition area> not selected is deleted, and the <transition option area> is deleted and replaced by the selected <transition area>.

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