International Telecommunication Union

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Z.109
(06/2007)

SERIES Z: LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

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

SDL-2000 combined with UML

ITU-T Recommendation Z.109

ITU-T

International Telecommunication Union

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
Extended Object Definition Language (eODL)	Z.130–Z.139
Testing and Test Control Notation (TTCN)	Z.140–Z.149
User Requirements Notation (URN)	Z.150–Z.159
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.

ITU-T Recommendation Z.109

SDL-2000 combined with UML

Summary

Objective: ITU-T Recommendation Z.109 defines a UML profile that maps to SDL-2000 semantics so that UML can be used in combination with SDL.

Coverage: This Recommendation presents a definition of the UML-to-SDL-2000 mapping for use in the combination of SDL-2000 and UML.

Application: The main area of application of this Recommendation is the specification of telecommunication systems. The combined use of SDL-2000 and UML permits a coherent way to specify the structure and behaviour of telecommunication systems, together with data.

Status/stability: This Recommendation is the complete reference manual describing the UML to SDL-2000 mapping for use in the combination of SDL-2000 and UML. It replaces the previous Rec. Z.109 that concerned an earlier version of UML and had a different style of profile description.

Associated work: ITU-T Recommendations Z.100, Z.104, Z.105, Z.106 and Z.107 concerning the ITU-T Specification and Description Language (SDL-2000).

Source

ITU-T Recommendation Z.109 was approved on 13 June 2007 by ITU-T Study Group 17 (2005-2008) under the ITU-T Recommendation A.8 procedure.

FOREWORD

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

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

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

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

NOTE

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

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

INTELLECTUAL PROPERTY RIGHTS

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

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

CONTENTS

Page

Introduction

The UML profile presented in this Recommendation is intended to support the usage of UML (version 2 or later) as a front-end for tools supporting specification and implementation of reactive systems, in particular for telecommunication applications. The intention is to enable tool vendors to create tools that benefit from the closure of semantic variations in UML with SDL-2000 semantics and benefit from SDL tool technology that supports this particular application area.

The intention is that when the profile is applied to a model, a set of stereotypes defined in this Recommendation extends the elements in the model and has several consequences:

– additional properties are available as specified by the stereotype attributes;
– constraints defined for the stereotypes apply to the model elements introducing more semantic checks that need to be fulfilled for the model;
– semantics, in particular dynamic semantics, are defined for the model elements as specified by the mapping of the stereotyped UML concepts to the SDL abstract grammar;
– a notation is given for language elements where no specific notation is provided by UML.

The details of the profile mechanism in this Recommendation follows: The Recommendation is structured in a number of clauses. Each clause defines one stereotype that captures the semantics of one SDL concept based on a UML concept. The stereotype in most cases constrains the UML element with a multiplicity of [1..1] (that is, the stereotype is required), but in some cases extends rather than constrains the basic UML language. The UML user never manually has to apply the stereotype to a UML element: instead stereotypes are applied automatically when applying the profile to the model itself, or if the user has not kept within the language defined by this profile a suitable message can be given to the user. As a consequence, applying this profile results in extra properties, extra semantic checks, and a well understood semantics that can be used in tools to provide features like static model analysis, simulation and application generation as the model is sufficiently well defined to be executable.

This Recommendation introduces notation for concepts that have no standard notation in UML, like a textual syntax for actions. To be able to uniquely define the mapping between the syntax and the model, the stereotypes introduced in these clauses extend the corresponding model element with multiplicity [0..1]. The idea is that when a user enters the described syntax, a tool should automatically create the corresponding model element with the correct stereotype applied.

ITU-T Recommendation Z.109

SDL-2000 combined with UML

1 Scope

This Recommendation defines a *UML profile for SDL-2000*. It ensures a well-defined mapping between parts of a UML model and the SDL-2000 semantics. The profile is based upon the UML meta-model and upon the abstract grammar of SDL, and in the following is referred to as SDL-UML.

The specializations and restrictions are defined in terms of the model elements of the UML meta-model and the abstract grammar of SDL and are in principle independent of notation. However, to generate the model elements in the UML meta-model a concrete notation will be used, and it is assumed that this notation is the notation defined by UML with the notation given in the Recommendation where UML allows options or gives no specific notation.

A software tool that claims to support this Recommendation (in the following called a tool) should be capable of creating, editing, presenting and analysing descriptions compliant with this Recommendation.

1.1 Conformance

A model that claims to be compliant to this Recommendation shall meet the meta-model constraints of UML and this Recommendation and, when mapped to the abstract grammar of SDL, shall conform to the abstract grammar of the Z.100 series of ITU-T Recommendations included by reference. A model is non-compliant if it does not meet the constraints or if it includes an abstract grammar that is not allowed by the Z.100 series of ITU-T Recommendations, or has analysable semantics that can be shown to differ from these Recommendations. Notation guidelines are given by this Recommendation, but (unlike SDL-2000 conformance) it is not essential for a model to be presented using the notation given in this Recommendation for it to conform to this Recommendation.

The abstract grammar of this Recommendation is a profile of UML and therefore any model that conforms to this Recommendation also conforms to the requirements of UML.

A tool that supports the profile shall support the specializations and restrictions of UML defined in the profile to conform to the Recommendation and should be capable of exporting such models to other tools and importing such models from other tools.

A conformance statement clearly identifying the profile features and requirements not supported should accompany any tool that handles a subset of this Recommendation. If no conformance statement is provided, it shall be assumed that the tool is fully compliant. It is therefore preferable to supply a conformance statement; otherwise, any unsupported feature allows the tool to be rejected as not valid. While it is suggested that tools follow notation guidelines, this is not a requirement. The issue of notation compliance is further considered in clause 1.2.

A **compliant tool** is a tool that is able to detect non-conformance of a model. If the tool handles a superset of SDL-UML, it is allowed to categorize non-conformance as a warning rather than a failure. It is required that for those 'Language Units' (see the UML specification [OMG UML] section 2, Conformance) handled by the tool, a compliant tool conforms to the abstract syntax defined by this profile combined with the UML specification [OMG UML] and the mapping of those 'Language Units' to the Z.100 abstract grammar defined by this Recommendation.

A **fully compliant tool** is a compliant tool that supports the complete profile defined by this Recommendation.

A **valid tool** is a compliant tool that supports a subset of the profile. A valid tool should include enough of the profile for useful modelling to be done. The subset shall enable the implementation of structured applications with communicating extended finite state machines.

1.2 Notation

This Recommendation gives notation guidelines for SDL-UML. For some of the UML elements, SDL-2000 contains elements that have a UML-like graphical notation, therefore it is preferable (but not mandatory) if SDL-2000 notation is used for these elements. A tool for the combined use of UML and SDL-2000 may use a *de facto* UML graphical notation standard for the SDL-UML covered by this Recommendation, but the SDL-2000 specific part of this tool should provide the graphical grammar for these elements as defined by [ITU-T Z.100].

Some UML notation is defined informally or by example in the UML specification [OMG UML], and the link between concrete notation and abstract grammar is not always well defined. Moreover, in some cases (in particular the syntax for actions and expressions), the UML specification [OMG UML] does not provide a complete language because no specific concrete notation is defined. In practice, the tool that is used defines the actual notation supported. This Recommendation provides a notation guideline to overcome this issue, by constraining notation in some cases where the UML specification [OMG UML] has options, and by defining notation in those cases where the UML specification has no specific notation or where the UML meta-model is extended by the profile.

In principle, it should be possible to exchange models between tools based on abstract grammar, provided the tools involved support the same abstract grammar and a common means of interchanging the abstract grammar. However, evaluating whether a tool actually supports a specific abstract grammar is difficult without inspecting the internal structure of the tool, so that normally tools are evaluated by observing how the concrete notation is handled. If the concrete notation handled by different tools differs significantly, it will be difficult to compare the support of the abstract grammar. So to compare the abstract grammar of tools without inspecting the contruction of the tools is easy only if they support the same (or very similar) concrete notation or possibly if they share an interchange format.

Checking if a model conforms to an abstract grammar has similar problems. Normally, this is simply done by checking if the model conforms to the concrete notation implemented by a tool, and checking (or assuming) the tool supports the abstract grammar. To check that a model matches the abstract grammar is usually done by checking the model matches the concrete notation and assuming this is correctly mapped to the abstract grammar. If the concrete notation differs, there is a further issue of determining the mapping to the abstract grammar. For this reason, to validate a model, if the concrete notation differs from the one given in this Recommendation, the differences should preferably be minor and how to map to either the notation or the abstract grammar of this Recommendation shall be defined.

1.3 Restrictions on SDL-2000 and UML

There are no restrictions on SDL-2000. However, SDL-2000 is only partially covered by SDL-UML. For example, exceptions are not included in this profile.

A general restriction on SDL-UML is that only the meta-model elements defined in this profile ensure a one-to-one mapping. In a combined use of UML and SDL-2000, more parts of UML can be used, but the mapping of these cannot be guaranteed to work the same with different tools.

This profile focuses on the following chapters of the UML Superstructure specification:

– Classes;

– Composite Structures;

– Common Behaviours;

– Actions;
– Activities;
– State Machines.

Meta-model elements defined in these clauses are included in this profile, if they are specifically mentioned in this Recommendation. Any meta-model element of the UML Superstructure specification that is not mentioned in this Recommendation is not included in this profile. A meta-model element that is a generalization of one of the included meta-model elements (that is, it is inherited) is included as part of the definition of the included meta-model element. Other specializations of such a generalization are only included if they are also specifically mentioned. If an included meta-model element has a property that is allowed to be non-empty, the meta-model element for the property is included. However, if the property is constrained so that it is always empty, such a property is effectively deleted from the model and therefore does not imply the meta-model element for the property is included.

Meta-model elements introduced in the following clauses of the UML Superstructure specification are not included in this profile:

– Components;
– Deployments;
– Use Cases;
– Interactions;
– Auxiliary Constructs;
– Profiles.

1.4 Mapping

UML classes generally represent entity types of SDL. In most cases, the entity kind is represented by stereotypes. Where predefined model-elements or stereotypes or keywords exist in UML that have a similar meaning as in SDL, they have been used.

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 Z.100] ITU-T Recommendation Z.100 (2002), *Specification and Description Language (SDL)*.

[ITU-T Z.105] ITU-T Recommendation Z.105 (2003), *SDL combined with ASN.1 modules (SDL/ASN.1)*.

[ITU-T Z.106] ITU-T Recommendation Z.106 (2002), *Common interchange format for SDL.*

[ITU-T Z.119] ITU-T Recommendation Z.119 (2007), *Guidelines for UML profile design*.

[OMG UML] OMG Unified Modelling Language: *Superstructure version 2.1.1 formal/07-02-05*.

NOTE – This Recommendation references specific paragraphs of [ITU-T Z.100] and [OMG UML]. The specific paragraph references are only valid for the editions specifically referenced above. If a more recent edition of [ITU-T Z.100] or [OMG UML] is used, it is possible that the corresponding paragraph number or paragraph heading is different.

3 Definitions

For the purposes of this Recommendation, the terms and definitions given in [ITU-T Z.100] and the following apply, and any term defined below applies if it is also defined elsewhere:

3.1 compliant tool: A tool that is able to detect non-conformance of a model to the profile defined by this Recommendation.

3.2 direct container: A is the direct container of B (B is directly contained in A; A directly contains B), if A contains B and there is no intermediate C that contains B such that C is contained in A.

3.3 fully compliant tool: A compliant tool that supports the complete profile defined by this Recommendation.

3.4 valid tool: A compliant tool that supports a subset of the profile defined by this Recommendation where the subset enables the definition of models containing structured applications with communicating extended finite state machines.

4 Abbreviations and acronyms

This Recommendation uses the following abbreviations:

BNF	Backus-Naur Form of syntax description
SDL	Specification and Description Language
SDL-UML	The notation defined by the UML profile in this Recommendation
UML	Unified Modelling Language 2.0 (see reference [OMG UML])
UML-SS	OMG UML-2.0 Superstructure Specification (see reference [OMG UML])

5 Conventions, names and templates

This clause defines conventions used throughout the rest of this Recommendation and the general handling of name resolution and template expansion that apply for the whole meta-model.

5.1 Conventions

The conventions defined in [ITU-T Z.119] apply. For convenience, these conventions are repeated below. The convention for a term enclosed in << and >> is extended to allow SDL qualifiers to be used. The convention for a term enclosed in < and > is extended to allow SDL concrete syntax to be used. A convention on the meaning of terms in italics is added.

A term in this Recommendation is a sequence of printing characters usually being either an English word or a concatenation of English words that indicate the meaning of the term.

A term preceded by the word "stereotype" names a UML stereotype used for this profile, according to the stereotype concept defined in the UML Superstructure specification documentation (usually in a phrase "The stereotype X extends the metaclass X" where X is a term). If the multiplicity of the stereotype is [1..1], the stereotype is required (that is the derived attribute isRequired of the Extension association between the extended metaclass and the stereotype is true). If the multiplicity of the stereotype is [0..1], the stereotype is not required.

An underlined term refers to a UML term or a term defined by a stereotype of this profile. A term starting with a capital letter by convention is the name of a metaclass.

A term enclosed in << and >> as brackets refers to a stereotype described by this profile, except if preceded by "SDL". For example, SDL <<**package** Predefined>> is the SDL qualifier (for the SDL Predefined package).

A term in italic in a stereotype description refers to an SDL-2000 abstract syntax item.

A term in < > brackets refers to a syntax rule defined in this profile, except if preceded by "SDL". For example SDL <name> refers to the SDL syntax for a name, whereas otherwise <name> refers to the syntax defined in clause 5.2. The profile includes syntax rules defined in UML-SS, so that if no explicit definition of a term in < > brackets is given, the syntax rule defined in UML-SS applies. The following syntax rules from UML-SS are used:

<assignment specification>, <attr name>, <prop type>, <visibility>, <default>

A term in Courier font (such as `pReply` in clause 9.6, CallOperationAction) refers to some text that appears in the model either as written by a user or to represent some text created from the expansion of a shorthand notation (as outlined in clause 5.4, Transformation, below and in detail for the relevant construct).

5.1.1 References

UML-SS: 6.4 The UML Meta-model

UML-SS: 8.3.8 Stereotype (from Profiles)

5.2 Names and name resolution: NamedElement

The stereotype NamedElement extends the metaclass NamedElement with multiplicity [1..1].

NOTE – Names are resolved according to the UML name binding rules. However, there are constraints applied to names that are mapped to the SDL abstract syntax.

5.2.1 Attributes

No additional attributes are defined.

5.2.2 Constraints

Any item that inherits from NamedElement and maps to SDL abstract syntax requiring a *Name* shall have a name. Any such name shall have a non-empty String value of characters derived from the syntax as defined in the Notation clause below.

When a complete SDL-UML model is mapped to the SDL abstract syntax, no item shall have the same *Name* as another item of the same entity kind in the same defining context.

NOTE – It is always possible to modify a UML model to meet the above naming requirement by renaming elements that generate name clashes so that the UML model is a valid SDL-UML model for this profile.

A NamedElement shall have a visibility and qualifiedName.

The visibility of the NamedElement shall not be **package**.

The visibility of the NamedElement (or of any item derived from it) shall be **protected** or **private** only if the NamedElement is an operation (including a literal) of a data type.

5.2.3 Semantics

The characters of the String for a name are each of the characters of the <name> taken in order.

Whenever a *Name* is required in the SDL abstract syntax (usually for the definition of an item), the *Name* is mapped from the name of the appropriate item derived from NamedElement. Whenever an *Identifier* is required in the SDL abstract syntax (usually to identify to a defined item), the *Identifier* is mapped from the name of the appropriate item derived from NamedElement. The detail of these mappings is described in the following paragraphs.

When a name is mapped to a *Name*, the String value of the name is mapped to the *Token* and if two items have a distinct String value each item maps a different *Token*. If two items that have the same *Token* for their *Name*, they have the same String value for their name. If two items have the same

String value for their name, they have the same *Token* for their *Name*, except if two UML elements are distinguishable by some additional means (such as distinct signatures of operations with the same name and same type in the same namespace). In such exceptional cases, each name is mapped to a different unique *Token*.

When the SDL abstract syntax requires an *Identifier*, the String value of the qualifiedName is used. A qualifiedName is a derived attribute that allows the NamedElement to be identified in a hierarchy. The *Qualifier* of the *Identifier* is a *Path-item* list that specifies uniquely the defining context of the identified entity and is derived from the qualifiedName. Starting at the root of the hierarchy, each name and class pair of the containing namespaces is mapped to the corresponding qualifier (*Package-qualifier*, *Agent-qualifier*, etc.) and name (*Package-name*, *Agent-name*, etc. respectively) pair. This mapping excludes the name of the NamedElement itself, which maps to the *Name* of the *Identifier*.

NOTE 1 – In SDL the *Qualifier* is usually derived by name resolution and context, and *Identifier* can usually be represented in the concrete syntax by an SDL <name> and the SDL qualifier part of an SDL <identifier> is omitted. Even in cases where an SDL qualifier needs to be given, usually some parts of the SDL qualifier can be omitted, so that a full context does not have to be given. Similarly in UML, qualifiedName is usually derived, and is not given explicitly in the concrete syntax. Thus in both UML and SDL an item can usually be identified in the concrete syntax simply by a name, whereas in the meta-model and abstract syntax the item will be identifed by a qualifiedName and *Identifier* respectively.

NOTE 2 – The visibility of a Package contained in another Package or a Class or other entity contained in a Package is handled by name resolution.

5.2.4 Notation

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

NOTE 1 – The syntax given for <name> assumes a one-to-one mapping between a <name> and an SDL <name> that has the same *Token*. The characters allowed in an SDL <name> are defined by ITU-T Rec. T.50: uppercase letters A (Latin capital letter A) to Z (Latin capital letter Z); lowercase letters a (Latin small letter a) to z (Latin small letter z); decimal digits 0 (Digit zero) to 9 (Digit nine); full stop and underline. The above syntax for <name> does not allow a full stop and requires a name to include at least one underline (first 2 alternatives of <name>) or at least one <letter>. UML supports alphabets and characters other than the Latin alphabet in ITU-T Rec. T.50. If these characters are used in names in this profile, the corresponding SDL <name> cannot have the same character string. This does not prevent mapping name in an extended alphabet to a *Name*, because the T.50 characters do not occur in the abstract grammar. Because the notation is a guideline rather than being mandatory, it is permitted to extend the syntax of <name> for this case.

```
<word> ::=
            <alphanumeric>+

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

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

<uppercase letter> ::=
            A   | B   | C   | D   | E   | F   | G   | H   | I   | J   | K   | L   | M
        |   N   | O   | P   | Q   | R   | S   | T   | U   | V   | W   | X   | Y   | Z

<lowercase letter> ::=
            a   | b   | c   | d   | e   | f   | g   | h   | i   | j   | k   | l   | m
        |   n   | o   | p   | q   | r   | s   | t   | u   | v   | w   | x   | y   | z
```

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

When a <name> occurs in syntax that defines a name, the qualifiedName is derived from the defining context. Otherwise, a name shall be bound according to the UML name binding rules and if necessary the name is qualified by containing namespaces. The following syntax for <identifier> is therefore given for specifying optionally qualified names.

<identifier> ::=
[<containing namespaces>] <name>

<containing namespaces> ::=
[<name separator>] { <name> <name separator> }+

NOTE 2 – <name separator> is defined in clause 11, Lexical rules.

If the <name> of an <identifier> is not unique and is ambiguous in the context where the <identifier> occurs, it shall be disambiguated by adding <containing namespaces> that contains one or more <name>s. In the absence of an initial <name separator>, the right-most <name>s in the <containing namespaces> shall unambiguously identify a context where the <name> of the <identifier> is defined. If the context is identified unambiguously by the right-most <name>s in the <containing namespaces>, it is allowed to add further <name>s to further identify or fully identify the context. If the initial <name separator> is given, the left-most name shall be a name defined at the top level of the model.

5.2.5 References

SDL: 6.1 Lexical rules

SDL: 6.3 Visibility rules, names and identifiers

UML-SS: 7.3.33 NamedElement (from Kernel, Dependencies)

UML-SS: 17.4.3 String (from PrimitiveTypes)

5.3 Template handling

Template parameters shall be expanded according to UML expansion rules before application of this profile, and therefore are not treated by this profile. For example, the ownedTemplateSignature, templateBinding, owningParameter and templateParameter are expanded before mapping to SDL abstract syntax.

5.3.1 References

UML-SS: 17.5 Templates

5.4 Transformation

The SDL abstract syntax of a model is generated from the concrete syntax of the SDL-UML model by the following process.

The concrete syntax of the model is parsed according to the SDL-UML concrete grammar. Where the concrete grammar defines shorthand notations, these are expanded during the parsing process before the corresponding meta-model items are generated.

NOTE 1 – The transformation models that are applied to expand shorthand notations are intended to be the same as expanding the corresponding shorthand notation in SDL. For example, an SDL remote procedure call is expanded into an exchange of implicit signals, and an SDL-UML remote operation call is also expanded into an exchange of signals.

To check if the concrete grammar is completely valid requires uses of names to be resolved, but Names are resolved according the SDL-UML meta-model. The parsing of the concrete grammar cannot therefore be done in complete isolation to generating the meta-model. For this reason, generating meta-model elements from the concrete syntax has to be done in parallel with parsing the concrete syntax. If the concrete model does not conform to the concrete grammar (syntax and static conditions for the concrete syntax, including the use of names) of SDL-UML, the model is not valid.

NOTE 2 – There is a general assumption that notation given in UML-SS and its relationship with the UML-SS meta-model is well-defined, and usually is not supplemented in this Recommendation. It is considered that if this assumption is false, it is an issue for the UML-SS and not an issue for this Recommendation. Where notation is not given in UML-SS, it is defined in this Recommendation.

Apart from the issue of name resolution mentioned above, the meta-model is generated from the concrete model according to the relationship between the concrete grammar and the meta-model. If the model expressed as meta-model elements does not conform to the abstract grammar (meta-classes, associations and constraints) of SDL-UML, the model is not valid.

Conforming to the meta-model rules of SDL-UML is a necessary (but not sufficient) condition for a model to be a valid model.

The model expressed as SDL-UML meta-model elements is mapped to a model in the abstract grammar of SDL. The behaviour of the model is determined by the behaviour defined for the SDL abstract grammar. The static conditions of SDL are reflected in the constraints of the SDL-UML meta-model. However, if during interpretation of the model expressed in the abstract grammar of SDL, any dynamic condition of SDL is not met, the model is not valid.

6 Stereotype summary

The following table gives a summary of the stereotypes defined in this profile with the UML metaclass each stereotype extends.

Stereotype	Stereotyped metaclass
ActiveClass	Class
Activity	Activity
ActivityFinalNode	ActivityFinalNode
AddStructuralFeatureValueAction	AddStructuralFeatureValueAction
AddVariableValueAction	AddVariableValueAction
Break	OpaqueAction
CallOperationAction	CallOperationAction
Class	Class
ConditionalNode	ConditionalNode
Connector	Connector
Continue	OpaqueAction
CreateObjectAction	CreateObjectAction
DataType	DataType
Decision	ConditionalNode
Empty	OpaqueAction
Enumeration	Enumeration
Expression	Expression
ExpressionAction	OpaqueAction
FinalState	FinalState

Stereotype	Stereotyped metaclass
For	LoopNode
If	ConditionalNode
InstanceValue	InstanceValue
Interface	Interface
LiteralInteger	LiteralInteger
LiteralNull	LiteralNull
LiteralString	LiteralString
LiteralUnlimitedNatural	LiteralUnlimitedNatural
LoopNode	LoopNode
OpaqueAction	OpaqueAction
Operation	Operation
Package	Package
PassiveClass	Class
Port	Port
PrimitiveType	PrimitiveType
Property	Property
Pseudostate	Pseudostate
Region	Region
ResetAction	OpaqueAction
Return	ActivityFinalNode
SendSignalAction	SendSignalAction
SequenceNode	SequenceNode
SetAction	OpaqueAction
Signal	Signal
State	State
StateMachine	StateMachine
Stop	ActivityFinalNode
Timer	Signal
Transition	Transition
ValueSpecification	ValueSpecification
While	LoopNode

7 Structure

The stereotypes below define static structural aspects of an SDL-UML model.

The following packages from UML are included:

– Communications

– Constructs (from Infrastructure library)

– Dependencies

– Interfaces

– InternalStructures

– Kernel
– Ports
– PrimitiveTypes (from Infrastructure library)

The following metaclasses from UML are included:

– Class
– Connector
– DataType
– Enumeration
– Interface
– Operation
– Package
– Port
– PrimitiveType
– Property
– Signal
– Timer

The metaclass ValueSpecification is included in clause 10, ValueSpecification.

7.1 ActiveClass

The stereotype ActiveClass is a concrete subtype of the stereotype Class.

NOTE – The concept of an active class (a class with isActive true) is separated from passive class (a class with isActive false) to distinguish the classes for executable agents that map onto SDL agent types.

7.1.1 Attributes

Stereotype attributes:

– isConcurrent: Boolean defines the concurrency semantics of an active class. If isConcurrent is false, all contained instances execute interleaved. If isConcurrent is true, contained instances execute concurrently, provided they are not also contained in an instance for which isConcurrent is false.

7.1.2 Constraints

- An <<ActiveClass>> Class shall have isActive == true.
- If isConcurrent is false, isConcurrent of any contained instance shall be false.
- If the <<ActiveClass>> Class has a classifierBehavior, it shall be a StateMachine.
- If an <<ActiveClass>> Class has a classifierBehavior and it has a superClass that is another <<ActiveClass>> Class that also has a classifierBehavior, the StateMachine of the sub-class shall redefine the StateMachine of the superClass. The reason is that in SDL the state machines of agents automatically extend each other, whereas this is not the case in UML.
- An <<ActiveClass>> Class used as the type of a composite property object (of another <<ActiveClass>> Class) shall have isAbstract == false (that is a typebased agent in an agent type shall not be based on an abstract type).
- An ownedAttribute that has a type that is an <<ActiveClass>> Class and where aggregationKind == composite shall not have public visibility (an agent instance set cannot be made visible ouside the enclosing agent type).

- A nestedClassifier shall not have public visibility (an agent type, data type, interface type or signal definition cannot be made visible outside the enclosing agent type).
- An ownedConnector shall not have public visibility (a channel cannot be made visible outside the enclosing agent type that owns the channel).
- An ownedPort shall have public visibility (gates are visible ouside the enclosing agent type).
- An ownedBehavior shall not have public visibility (a procedure or composite state type cannot be made visible outside the enclosing agent type).
- An ownedBehavior shall only contain a StateMachine.

7.1.3 Semantics

An <<ActiveClass>> Class is mapped to an *Agent-type-definition*.

The name of the Class maps to the *Agent-type-name* of the *Agent-type-definition*.

The isConcurrent attribute maps to the *Agent-kind* of the *Agent-type-definition*. If isConcurrent == true, the *Agent-kind* is a *BLOCK*, otherwise (isConcurrent == false) the *Agent-kind* is a *PROCESS*.

NOTE 1 – The concurrency behaviour is that state machines within a *PROCESS* instance (for the instance itself and contained *PROCESS* instances) are interleaved, and agent instances directly contained within a *BLOCK* (even multiple instances of the same *PROCESS*) are logically concurrent. Actual concurrency depends on implementation constraints such as the number of execution engines.

The qualifiedName of the optional general property (and thus the generalization property and the derived property superClass) maps to the *Agent-type-identifier* of the *Agent-type-definition* that represents inheritance in the SDL abstract syntax.

The nestedClassifier, ownedAttribute, ownedConnector, ownedPort and ownedOperation associations map to the rest of the contents of the *Agent-type-definition* as described below.

A nestedClassifier that is an <<ActiveClass>> Class maps to an element of the *Agent-type-definition-**set*** of the *Agent-type-definition*.

A nestedClassifier that is a <<PassiveClass>> Class maps to an *Object-data-type-definition* that is an element of the *Data-type-definition-**set*** of the *Agent-type-definition*.

A nestedClassifier that is DataType maps to a *Value-data-type-definition* that is an element of the *Data-type-definition-**set*** of the *Agent-type-definition*.

A nestedClassifier that is an Interface maps to an *Interface-type-definition* that is an element of the *Data-type-definition-**set*** of the *Agent-type-definition*.

A nestedClassifier that is a Signal maps to a *Signal-definition* that is an element of the *Signal-definition-**set*** of the *Agent-type-definition*.

An ownedAttribute is a Property. The mapping defined in clause 7.12, Property, applies.

An ownedAttribute that maps to a *Variable-definition* (see clause 7.12, Property) is an element of the *Variable-definition-**set*** of the *Agent-type-definition*. An ownedAttribute that is visible outside the <<ActiveClass>> Class (public visibility) and that has a type that is a DataType or <<PassiveClass>> Class is the *Variable-definition* for an exported variable and also maps to an implicit *Signal-definition* pair for accessing this exported variable in the defining context of the *Agent-type-definition*.

An ownedAttribute that maps to an *Agent-definition* (see clause 7.12, Property) is an element of the *Agent-definition-**set*** of the *Agent-type-definition*.

Each Connector of the ownedConnector maps to an element of the *Channel-definition-**set*** of the *Agent-type-definition*.

Each Port of the ownedPort maps to an element of the *Gate-definition-**set*** of the *Agent-type-definition*.

Each Behavior of the ownedBehavior maps to an element of either the *Composite-state-type-definition-**set*** or the *Procedure-definition-**set***. If the owned Behavior is the method of an Operation, it is an element of the *Procedure-definition-**set***, otherwise it is an element of the *Composite-state-type-definition-**set***. The StateMachine that is the Behavior of the optional classifierBehavior maps to the *State-machine-definition* of the *Agent-type-definition* (see clause 8.5, StateMachine). The name of the optional classifierBehavior is mapped to the *State-name* of the *State-machine-definition*. The *Composite-state-type-identifier* of this *State-machine-definition* identifies the *Composite-state-type* derived from the StateMachine that is the classifierBehavior.

NOTE 2 – The UML StateMachine maps to the behaviour of an SDL composite state type, and the *State-machine-definition* references this behaviour.

The ownedParameter set of the Behavior of classifierBehavior maps to the *Agent-formal-parameter* list of the *Agent-type-definition*.

NOTE 3 – It is a semantic variation in UML-SS whether one or more behaviours are triggered when an event satisfies multiple outstanding triggers.

NOTE 4 – It is currently not allowed to give actual parameter value to a formal parameter of an agent (see clause 9.9, CreateObjectAction).

An event satisfies only one trigger (a signal can initiate only one input transition).

NOTE 5 – In UML-SS, ordering of the events in the input pool and therefore the selection of the next event to be considered is a semantic variation.

At any specific wait point (that is, in a specific state), events for a trigger of higher priority are considered before those of triggers of lower priority. Within a given trigger priority, the events in the input pool are considered in the order of arrival in the input pool, therefore if all triggers have the same priority, the events are considered in order of arrival. If an event in the input pool of events satisfies no triggers at a wait point, it is left in the input pool if it is deferred at that wait point, or (if it is not deferred) it is consumed triggering an empty transition leading to the same wait point.

7.1.4 Notation

The UML presentation option for active class (see Figure 13.14 – Active class in UML-SS) is used, and defines that an active class shall be shown using an extra vertical bar on either side.

7.1.5 References

SDL:	8.1.1	Structural type definitions
	8.2	Context parameters
	8.3	Specialization
	8.4	Type references
UML-SS:	7.3.6	BehavioredClassifier (from Interfaces)
	7.3.7	Class (from Kernel)
	9.3.1	Class (from StructuredClasses)
	9.3.8	EncapsulatedClassifier (from Ports)
	13.3.2	Behavior (from BasicBehaviors)
	13.3.4	BehavioredClassifier (from BasicBehaviors, Communications)
	13.3.8	Class (from Communications)

7.2 Class

The stereotype Class extends the merged metaclass Class with multiplicity [1..1]. The stereotype Class is abstract.

NOTE 1 – SDL-UML separates the merged metaclass Class into ActiveClass for components of a system that have behaviour based on state machines, and PassiveClass for data that is contained within the state machines or passed as information between state machines. ActiveClass and PassiveClass are key stereotypes in this profile.

NOTE 2 – In UML-SS, the metaclass Class is the merger of Class (from Kernel), the Class (from StructuredClasses) and the Class (from Communications). The metaclass Class (from StructuredClasses) extends the UML metaclass Class (from Kernel) with the capability to have an internal structure and ports. The metaclass Class (from Communications) is a specialization of BehavioredClassifier and a merge of Class (from Kernel). The UML metaclass Class (from Communications) is either active (each of its instances having its own thread of control) or passive (its instances executing within the context of some other object).

NOTE 3 – The merged metaclass Class has following properties because of the specializations from other metaclasses: classifierBehavior in Class (from Communications) from BehavioredClassifier (from BasicBehaviors, Communications); interfaceRealization in Class (from Communications) from BehavioredClassifier (from Interfaces); ownedBehavior in Class (from Communications) from BehavioredClassifier (from BasicBehaviors, Communications); ownedConnector in Class (from StructuredClasses) from EncapsulatedClassifier from StructuredClassifier; ownedParameter in a Class (from Kernel) that is specialized as a Behavior; ownedPort in Class (from StructuredClasses) from EncapsulatedClassifier; ownedReception from Class (from Communications); ownedTrigger in Class (from Communications) from BehavioredClassifier (from BasicBehaviors, Communications).

7.2.1 Attributes

No additional attributes.

7.2.2 Constraints

- Every <<Class>> Class shall be either an <<ActiveClass>> Class or a <<PassiveClass>> Class.
- Multiple inheritances are not allowed, so there shall be at most one element in the generalization property of the <<Class>> Class.
- A <<Class>> Class that redefines another Class (as specified by redefinedClassifier) shall have the same name as the redefined Class, because in SDL redefined types have the same name as the original type.
- The clientDependency shall not include an InterfaceRealization, because interfaces are not realized directly but only via ports.

7.2.3 Semantics

The concrete subtypes of the stereotype Class give its semantics.

7.2.4 Notation

The concrete subtypes of the stereotype Class give its notation.

7.2.5 References

UML-SS: 7.3.6 BehavioredClassifier (from Interfaces)

7.3.7 Class (from Kernel)

9.3.1 Class (from StructuredClasses)

9.3.8 EncapsulatedClassifier (from Ports)

13.3.2 Behavior (from BasicBehaviors)

13.3.4 BehavioredClassifier (from BasicBehaviors, Communications)

13.3.8 Class (from Communications)

7.3 Connector

The stereotype Connector extends the metaclass Connector with multiplicity [1..1].

NOTE – In UML-SS, connector is a general concept for a communication link between two instances and the mechanism for communication could be by parameter passing in variables or slots, via pointers or some other means. In this profile connectors only provide communication by signals, which are identified by the information flows associated with the connector and the connector maps to a *Channel-definition*.

7.3.1 Attributes

Stereotype attributes:

– delay: Boolean If true the signals transported on the connector are delayed. The default value is true.

7.3.2 Constraints

- In the case of an InformationItem associated with an InformationFlow associated with a Connector, the represented property of the InformationItem shall be a Signal or an Operation or an Interface.
- There shall always be exactly 2 end properties.
- A ConnectorEnd that is part of the end property shall have empty lowerValue and upperValue properties.
- The role property of a ConnectorEnd that is part of the end property of the Connector shall be a Port.
- The type property shall be empty.
- The redefinedConnector property shall be empty.
- The isStatic property shall be false.
- There shall be at least one InformationFlow associated with a Connector.

7.3.3 Semantics

A <<Connector>> Connector maps to a *Channel-definition*.

The name attribute defines the *Channel-name*.

An InformationFlow associated with a <<Connector>> Connector defines the *Signal-identifier-**set*** of a *Channel-path* as follows. The conveyed InformationItem set of each InformationFlow defines the *Signal-identifier-**set*** of the *Channel-path*. If the InformationItem set is omitted, then the *Signal-identifier-**set*** is computed based on the realized and required interface of the attached Port. If the InformationFlow conveys an Interface, then the *Signal-identifier-**set*** is computed according to the transformation rules of Z.100 (see clause 7.6, Interface).

InformationFlow in one direction only (with or without InformationItems) implies that the channel is unidirectional.

InformationFlow in both directions (with or without InformationItems) implies that the channel is bidirectional.

If the delay attribute is false, this maps to *NODELAY*. If the delay attribute is true, *NODELAY* is omitted.

The end property defines the gates of each *Channel-path* as follows.

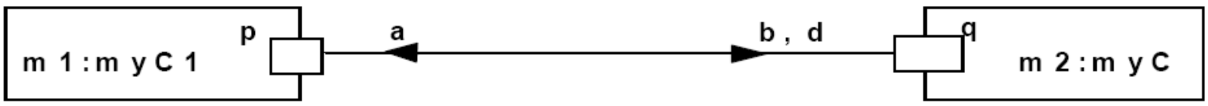
The role of a ConnectorEnd that is part of the end property maps to an *Originating-gate* or *Destination-gate* in each *Channel-path*. If the role corresponds to the source of the InformationFlow for the *Channel-path*, the role maps to an *Originating-gate*, otherwise, it maps to a *Destination-gate*. The *Gate-identifier* is derived from the name of the Port given by the role.

If the partWithPort is non-empty, *Gate-identifier* contains as its last path-name (before the name of the gate) the name of the part identified with partWithPort.

7.3.4 Notation

No additional notation, but additional meaning is given to the position of arrows on connectors. The notation in UML-SS 9.3.6 is used together with the notation for information flows on connectors as described in UML-SS 17.2.

An example:



If delay is false for the connector, the information flow arrows are at the end of the connector lines. If true, the arrows are as shown in the figure above.

7.3.5 References

SDL:	10.1	Channel
UML-SS:	9.3.6	Connector (from InternalStructures)
	9.3.7	ConnectorEnd (from InternalStructures, Ports)
	17.2	InformationFlows (from InformationFlows)

7.4 DataType

The stereotype DataType extends the metaclass DataType with multiplicity [1..1].

NOTE – A <<DataType>> Datatype is a PrimitiveType (which captures the non-parameterized predefined data types of SDL) or an Enumeration (which corresponds to types defined by a set of literal names) or a value type (typically a structured data type, but could be simply a collection of operations). If it is a value type with at least one ownedAttribute, it is a value structure type (see also the definition of an object structure type by a <<PassiveClass>> Class with an ownedAttribute set that is not empty). A value type without an ownedAttribute that is neither a PrimitiveType nor an Enumeration is a collection of operations. If some of these operations have a result of the type, these denote values of the type. For example, the basis of a type for imaginary numbers could be a type called Imaginary with an operation `makeImaginary(Integer,Integer)->Imaginary` together with other appropriate operations and `makeImaginary(-1,2)` would denote a value of the type.

7.4.1 Attributes

No additional attributes.

7.4.2 Constraints

- An ownedAttribute shall have a type that is a <<PassiveClass>> Class or a Datatype.

  NOTE – A data type cannot contain agent instance sets or variables. A data type with an ownedAttribute is a value structure type and each ownedAttribute is field of the structure.
- interfaceRealization shall be empty.
- There shall be at most one element in the generalization property of the type, as multiple inheritances are not allowed in SDL.
- Each ownedOperation association shall specify one of the operations defined for the specific data type.

- The name of a <<DataType>> Datatype that is not a PrimitiveType shall not be one of the names required for a <<PrimitiveType>> PrimitiveType.

7.4.3 Semantics

The following semantics only apply if the <<Datatype>> Datatype is neither a <<PrimitiveType>> PrimitiveType nor an <<Enumeration>> Enumeration.

A <<Datatype>> Datatype is mapped to a *Value-data-type-definition*.

Every <<Datatype>> Datatype has an operation for equality in the SDL-UML model that maps to the SDL operation `equal` for the *Value-data-type-definition*. If an explicit definition of the equality operation is given, this is used; otherwise, the operation is an implicit operation that matches the default SDL semantics for `equal`.

The name of the <<Datatype>> Datatype maps to the *Sort* of the *Value-data-type-definition*.

The qualifiedName of the optional general, if present, maps to the *Data-type-identifier* of the *Value-data-type-definition* that represents inheritance in the SDL abstract syntax.

The ownedOperation items are mapped to the *Static-operation-signature-**set*** of the *Value-data-type-definition*.

An ownedBehavior maps to a *Procedure-definition* in the *Procedure-definition-**set*** in the nearest enclosing scope that contains the *Value-data-type-definition*.

A <<Datatype>> Datatype with an ownedAttribute set that is not empty represents a structure and each ownedAttribute represents a field. An ownedAttribute maps to operations in the *Static operation-signature-**set*** in the SDL abstract syntax for the field operations. These operations are determined and corresponding items implied in the SDL-UML model in the same way as the field operations for a <<PassiveClass>> Class with an ownedAttribute set that is not empty (see clause 7.9, PassiveClass). If ownedOperation associations are defined, the defined operation signatures are added to the *Static-operation-signature-**set***. The contained *Data-type-definition-**set***, *Syntype-definition-**set*** and *Exception-definition-**set*** are empty.

7.4.4 Notation

UML standard syntax is used.

7.4.5 References

SDL:	12.1	Data definitions
	12.1.2	Data type definition
	12.1.9.4	Syntypes
	12.1.3	Specialization of data types
	12.1.7.2	Structure data types
UML-SS:	7.3.11	DataType (from Kernel)

7.5 Enumeration

The stereotype Enumeration extends the metaclass Enumeration with multiplicity [1..1].

NOTE – An enumeration is a data type that has values that are user defined literals and is simply mapped to an SDL data type that has literals defined.

7.5.1 Attributes

No additional attributes.

7.5.2 Constraints

- The ownedAttribute set shall be empty.

NOTE – An enumeration is allowed to be a structured data type.

7.5.3 Semantics

An <<Enumeration>> Enumeration is mapped to a *Value-data-type-definition*.

The name maps to the *Sort* of the *Value-data-type-definition*.

The generalization property maps to the *Data-type-identifier* that represents inheritance in the SDL abstract syntax.

The ownedLiteral property maps to the *Literal-signature-**set***.

Each ownedOperation association maps to an element of the *Static-operation-signature-**set***. The contained *Data-type-definition-**set***, *Syntype-definition-**set*** and *Exception-definition-**set*** are empty.

7.5.4 Notation

UML standard syntax is used.

7.5.5 References

SDL:	12.1	Data definitions
	12.1.2	Data type definition
	12.1.3	Specialization of data types
	12.1.7.1	Literals
UML-SS:	7.3.16	Enumeration (from Kernel)

7.6 Interface

The stereotype Interface extends the metaclass Interface with multiplicity [1..1].

NOTE – An interface defines public features that are used to communicate with an object. In SDL-UML, these are signals, remote variables and remote procedures. Accesses to remote variables and calls of remote procedures are signal exchanges in the SDL abstract grammar, so the components of a SDL-UML interface map to signals in the corresponding *Interface-definition*.

7.6.1 Attributes

No additional attributes.

7.6.2 Constraints

- Each nestedClassifier shall be a Signal.
- The ownedReception property shall be empty.

7.6.3 Semantics

A <<Interface>> Interface is mapped to an *Interface-definition*.

The name defines the *Sort* of the *Interface-definition*.

The general property defines the *Data-type-identifier* list that represents inheritance in the SDL abstract syntax. Each general property shall be an Interface.

The nestedClassifier, ownedAttribute, ownedOperation properties define the rest of the contents of the interface.

The ownedAttribute, ownedOperation properties are transformed to signals according to the SDL rules for remote variables (see clause 10.6 of [ITU-T Z.100]) and remote procedures (see clause 10.5 of [ITU-T Z.100]) and are thus mapped to *Signal*s in the *Signal-definition-**set*** of the *Interface-definition*.

Each nestedClassifier property (each of which is a Signal, see constraints above) maps to an element of the *Signal-definition-**set*** of the *Interface-definition*.

7.6.4 Notation

UML standard syntax is used.

7.6.5 References

SDL:	12.1	Data definitions
	10.5	Remote procedures
	10.6	Remote variables
UML-SS:	7.3.24	Interface (from Interfaces)
	13.3.15	Interface (from Communications)

7.7 Operation

The stereotype Operation extends the metaclass Operation with multiplicity [1..1].

NOTE – An operation is a feature that determines how an object behaves as described by its method. If the operation is contained in an agent (that is an <<ActiveClass>> Class), the method has to be a state machine and maps to a procedure. An operation contained in an interface is treated as a remote procedure. Otherwise, the operation has to be an activity and maps to an operation of the SDL data type for the <<PassiveClass>> Class or <<DataType>> DataType that contains the operation.

7.7.1 Attributes

No additional attributes.

7.7.2 Constraints

- If the class of an <<Operation>> Operation is a <<PassiveClass>> Class or <<DataType>> DataType, the <<Operation>> Operation shall not be generalized: that is, there shall not be an operation that inherits from an operation defined in a passive class or data type, and the method associated with the <<Operation>> Operation shall be an Activity.
- If the class of an <<Operation>> Operation is an <<ActiveClass>> Class, the method associated with the <<Operation>> Operation shall be a StateMachine.
- Both the <<Operation>> Operation and the dynamically corresponding method shall be defined in the same Class.
- The ownedParameter set of the <<Operation>> Operation shall be the same as the ownedParameter set of the method implementing the operation.
- The raisedException shall be empty.
- The name and ownedParameter set of the <<Operation>> Operation shall be the same as the name and ownedParameter set of any redefined operation (as given by the redefinedOperation property).

7.7.3 Semantics

An <<Operation>> Operation directly contained in an <<ActiveClass>> Class is mapped to a *Procedure-definition*. The name defines the *Procedure-name*. The rest of the *Procedure-definition* is defined as described below.

An <<Operation>> Operation directly contained in a <<PassiveClass>> Class or <<DataType>> DataType is mapped to an *Operation-signature* and an anonymous *Procedure-definition* identified by the *Identifier* in the abstract syntax for the *Operation-signature*. The *Procedure-definition* is placed in the same context as the data type corresponding to the <<PassiveClass>> Class or <<DataType>> DataType. The rest of the *Procedure-definition* is defined as described below. The name defines the *Operation-name* of the *Operation-signature*. In each ownedParameter that does not have a return direction, the type and multiplicity together define (in order of the parameters) a *Formal-argument* of the *Operation-signature* with a type determined in the same way as in a <<Property>> Property (see clause 7.12, Property). The type of the <<Operation>> Operation defines the *Result* of the *Operation-signature*.

NOTE 1 – The type of the <<Operation>> Operation is derived from the ownedParameter that has a return direction.

An <<Operation>> Operation contained in an Interface is mapped to signals according to the rules described in clause 7.6 for Interface semantics.

If the <<Operation>> Operation maps to a *Procedure-definition* (named or anonymous), each ownedParameter that does not have a return direction defines (in order) a *Procedure-formal-parameter* where the name and type (including the multiplicity) of the ownedParameter define respectively the *Variable-name* and the *Sort-reference-identifier* of the *Parameter*. The *Sort-reference-identifier* is determined in the same way as for a <<Property>> Property (see clause 7.12, Property). The direction (in, inout, or out) of each ownedParameter that does not have a return direction determines (respectively) if the corresponding *Procedure-formal-parameter* is an *In-parameter* or *Inout-parameter* or *Out-parameter*. The type of the <<Operation>> Operation defines the *Result* of the *Procedure-definition*. The Behavior identified by the method property defines the *Procedure-graph*, *Data-type-definition-**set***, and *Variable-definition-**set*** of the *Procedure-definition*. The general property (derived from generalization) maps to the optional *Procedure-identifier* that is part of the *Procedure-definition* and identifies the inherited procedure (if any).

The following properties of an Operation are ignored when mapping to SDL:

– isQuery
– bodyCondition
– precondition
– postcondition

NOTE 2 – In UML-SS, an operation cannot itself directly contain an operation, so that when the model is mapped to the Z.100 abstract syntax, there will never be a procedure contained within a procedure (that is a local procedure).

7.7.4 Notation

UML standard syntax is used.

7.7.5 References

SDL:	9.5	Procedure
	12.1.4	Operations
	10.5	Remote procedures
	10.6	Remote variables
UML-SS:	7.3.5	BehavioralFeature (from Interfaces)
	7.3.36	Operation (from Kernel, Interfaces)
	13.3.3	BehavioralFeature (from BasicBehaviors, Communications)
	13.3.22	Operation (from Communications)

7.8 Package

The stereotype Package extends the metaclass Package with multiplicity [1..1].

NOTE – The concept of a package in UML is simply mapped to a package in SDL.

7.8.1 Attributes

No additional attributes are defined.

7.8.2 Constraints

- All ownedMember elements of the Package shall belong to items for which mappings or transformations are described in this profile.
- The packageMerge composition shall be empty.
- The name of the Package shall not be empty.

NOTE – ownedTemplateSignature and templateBinding should always be empty after template expansion.

7.8.3 Semantics

A <<Package>> Package is mapped to a *Package-definition*.

The name of the package maps to the *Package-name* of the *Package-definition*.

The elements of the ownedMember composition define the contents of the package, that is the *Package-definition-**set***, *Data-type-definition-**set***, *Syntype-definition-**set***, *Signal-definition-**set***, *Agent-type-definition-**set***, *Composite-state-type-definition-**set*** and *Procedure-definition-**set***. Each ownedMember that is a nestedPackage maps to an element of the *Package-definition-**set*** of the *Package-definition*. An ownedMember that is not a nestedPackage is mapped as defined in other sections to a *Data-type-definition*, *Syntype-definition*, *Signal-definition*, *Agent-type-definition*, *Composite-state-type-definition* or *Procedure-definition* element of the corresponding set of the *Package-definition*.

NOTE – The UML ElementImport and PackageImport (which are not stereotyped in this profile) define the import and visibility of elements of the package and define the name resolution of imported package elements. The resolved items map to *Name* and *Identifier* items in the SDL abstract syntax as described in clause 5.2.

7.8.4 Notation

UML standard syntax is used.

7.8.5 References

SDL:	7.2	Package
UML-SS:	7.3.37	Package (from Kernel)

7.9 PassiveClass

The stereotype PassiveClass is a concrete subtype of the stereotype Class.

NOTE – The concept of a passive class (a class with isActive false) is separated from active class (a class with isActive true) to distinguish the classes for reference types that map onto object data types in SDL.

7.9.1 Attributes

No additional attributes are defined.

7.9.2 Constraints

- A <<PassiveClass>> Class shall have isActive == false.
- A <<PassiveClass>> Class shall have no classifierBehavior.

- A nestedClassifier shall be a <<PassiveClass>> Class or a Datatype (which includes PrimitiveType or EnumerationType) or an Interface.
- An ownedAttribute where aggregation == composite shall have a type that is a <<PassiveClass>> Class or a Datatype.
- The ownedConnector shall be empty.
- The ownedPort shall be empty.
- The ownedTrigger shall be empty.
- Each ownedBehavior shall be a StateMachine that does not contain State elements and has an Activity that only contains one SequenceNode.
- The ownedReception should be empty.

7.9.3 Semantics

A <<PassiveClass>> Class is mapped to an *Object-data-type-definition*.

The name of the <<PassiveClass>> Class maps to the *Sort*.

The qualifiedName of the optional general if present (and thus the generalization property and the derived property superClass) maps to the *Data-type-identifier* of the *Object-data-type-definition* that represents inheritance in the SDL abstract syntax.

The nestedClassifier, ownedAttribute and ownedOperation associations map to the rest of the contents of the *Object-data-type-definition* as described below.

A nestedClassifier that is a <<PassiveClass>> Class maps to an *Object-data-type-definition* that is an element of the *Data-type-definition-**set*** of the *Data-type-definition*.

A nestedClassifier that is a Datatype (which includes EnumerationType or PrimitiveType) maps to a *Value-data-type-definition* that is an element of the *Data-type-definition-**set*** of the *Data-type-definition*.

A nestedClassifier that is an Interface maps to an *Interface-definition* that is an element of the *Data-type-definition-**set*** of the *Data-type-definition*.

A <<PassiveClass>> Class with an ownedAttribute set that is not empty represents a structure and each ownedAttribute represents a field. An ownedAttribute maps to operations in the *Dynamic-operation-signature-**set*** in the SDL abstract syntax for the field operations as described in [ITU-T Z.100]. The operations are equivalent to supporting the following methods:

– Make (field-sort-list) -> S;

– **virtual** field-modify-operation-name (field-sort) -> S;

– **virtual** field-extract-operation-name -> field-sort;

– field-presence-operation-name -> Boolean;

where:

S is the qualifiedName of the <<PassiveClass>> Class, field-sort-list is each field-sort (see below) listed in order of the ownedAttribute list, field-sort is the qualifiedName of the type of the ownedAttribute, field-modify-operation-name is the name of the ownedAttribute concatenated with "Modify", field-extract-operation-name is the name of the ownedAttribute concatenated with "Extract", field-presence-operation-name is the name of the ownedAttribute concatenated with "Present", and **virtual** denotes the method can be redefined if the <<PassiveClass>> Class is specialized. The corresponding items are implied in the SDL-UML model so that the operations are valid in expressions.

An ownedBehavior maps to a *Procedure-definition* in the *Procedure-definition-**set*** in the nearest enclosing scope that contains the *Object-data-type-definition*.

The ownedOperation items are mapped to items in the *Dynamic-operation-signature-**set***. The implicit parameter corresponding to the containing class is considered virtual, the rest of the parameters and the return type are non-virtual.

7.9.4 Notation

UML standard class syntax is used.

7.9.5 References

SDL:	8.2	Context parameters
	8.3	Specialization
	8.4	Type references
	12.1.2	Data type definition
	12.1.3	Specialization of data types
	12.1.7.2	Structure data types
UML-SS:	7.3.6	BehavioredClassifier (from Interfaces)
	7.3.7	Class (from Kernel)
	9.3.1	Class (from StructuredClasses)
	9.3.8	EncapsulatedClassifier (from Ports)
	13.3.2	Behavior (from BasicBehaviors)
	13.3.4	BehavioredClassifier (from BasicBehaviors, Communications)
	13.3.8	Class (from Communications)

7.10 Port

The stereotype Port extends the metaclass Port with multiplicity [1..1].

NOTE – An SDL-UML port defines an SDL *Gate*. The required interfaces characterize the requests from the classifier to its environment through the port and therefore define the outgoing signals for the *Gate*. The provided interfaces of a port characterize requests to the classifier that are permitted through the port and therefore define the incoming signals for the *Gate*.

7.10.1 Attributes

No additional attributes.

7.10.2 Constraints

The <<Port>> Port referenced by redefinedPort shall have the same name as the current Port.

The aggregationKind shall be composite.

The isDerived and isDerivedUnion properties shall be false.

The isReadOnly property shall be true.

The defaultValue property shall be empty.

The subsettedProperty property shall be empty.

The qualifier property shall be empty.

The isStatic property shall be false.

The lowerValue and upperValue properties shall be ValueSpecifications that evaluate to 1.

The isService property shall be false.

7.10.3 Semantics

A <<Port>> Port is mapped to a *Gate-definition*.

The name defines the *Gate-name*.

The requiredInterface property maps to the *Out-signal-identifier-**set***. The set is computed according to the rules given in clause 12.1.2 of [ITU-T Z.100].

The providedInterface property defines the *In-signal-identifier-**set***. The set is computed according to the rules given in clause 12.1.2 of [ITU-T Z.100].

If isBehavior is true, a channel is constructed in the SDL abstract syntax that connects the gate and the state machine of the containing agent.

7.10.4 Notation

UML standard syntax is used.

7.10.5 References

SDL: 8.1.5 Gate

UML-SS: 9.3.11 Port (from Ports)

7.11 PrimitiveType

The stereotype PrimitiveType extends the metaclass PrimitiveType with multiplicity [1..1].

NOTE – A primitive type defines a predefined data type. For SDL-UML these are the predefined data items of SDL, or local definitions that specialize the data item.

7.11.1 Attributes

No additional attributes.

7.11.2 Constraints

- The ownedAttribute set shall be empty.
- The name shall be one of the following: `Boolean`, `Integer`, `UnlimitedNatural`, `Character`, `Charstring`, `Real`, `Duration`, `Time`, `Bit`, `Bitstring`, `Octet`, `Octetstring` or `Pid`.
- The generalization property shall be empty.
- Each ownedOperation association shall specify one of the operations defined for the specific data type (see clause 12, Predefined Data, clauses 12.1.6 and D.3 of [ITU-T Z.100]).

7.11.3 Semantics

If no ownedOperation associations are defined, each <<PrimitiveType>> PrimitiveType is mapped to a predefined *Syntype-definition* or a predefined *Value-data-type-definition* as detailed in the next paragraph. All the contents of the *Value-data-type-definition* (such as the *Literal-signature-**set***) are implied from the mapping to the specific definition of the SDL `<<`**`package`**` Predefined>>` item as further defined in clause 12.1. The corresponding items (such as ownedBehavior for the operations) are implied in the SDL-UML meta-model and therefore can be used in expressions.

The name `UnlimitedNatural` maps to the *Syntype-name* for

SDL `<<`**`package`**` Predefined>> Natural` *Syntype-definition*, otherwise the name maps to the *Sort* of the *Value-data-type-definition* as follows:

`Array` maps to the *Sort* for SDL `<<package Predefined>> Array`.

`Bag` maps to the *Sort* for SDL `<<package Predefined>> Bag`.

`Bit` maps to the *Sort* for SDL `<<package Predefined>> Bit`.

`Bitstring` maps to the *Sort* for SDL `<<package Predefined>> Bitstring`.

`Boolean` maps to the *Sort* for SDL `<<package Predefined>> Boolean`.

`Character` maps to the *Sort* for SDL `<<package Predefined>> Character`.

`Charstring` maps to the *Sort* for SDL `<<package Predefined>> Charstring`.

`Duration` maps to the *Sort* for SDL `<<package Predefined>> Duration`.

`Integer` maps to the *Sort* for SDL `<<package Predefined>> Integer`.

`Octet` maps to the *Sort* for SDL `<<package Predefined>> Octet`.

`Octetstring` maps to the *Sort* for SDL `<<package Predefined>> Octetstring`.

`Pid` maps to the *Sort* for SDL `<<package Predefined>> Pid`.

`Powerset` maps to the *Sort* for SDL `<<package Predefined>> Powerset`.

`Real` maps to the *Sort* for SDL `<<package Predefined>> Real`.

`String` maps to the *Sort* for SDL `<<package Predefined>> String`.

`Time` maps to the *Sort* for SDL `<<package Predefined>> Time`.

`Vector` maps to the *Sort* for SDL `<<package Predefined>> Vector`.

If ownedOperation associations are defined, the <<PrimitiveType>> PrimitiveType is mapped to a *Value-data-type-definition* that has a *Data-type-identifier* for the inherited SDL `<<package Predefined>>` item that has the *Sort* for the name as defined above, or for `UnlimitedNatural` a *Syntype-definition* with the *Parent-sort-identifier* `Integer` and a *Value-data-type-definition* that has a *Data-type-identifier* for SDL `<<package Predefined>> Integer`. Therefore if ownedOperation associations are defined, a local item is introduced that inherits from the item of the same name in SDL `<<package Predefined>>` and adds the defined operation signatures to the *Static-operation-signature-**set***. The contained *Data-type-definition-**set***, *Syntype-definition-**set*** and *Exception-definition-**set*** are empty.

7.11.4 Notation

UML standard syntax is used.

7.11.5 References

SDL:	12.1	Data definitions
	12.1.2	Data type definition
	12.1.6	Pid and pid sorts
	12.1.9.4	Syntypes
	D.3	Package Predefined
UML-SS:	7.3.43	PrimitiveType (from Kernel)

7.12 Property

The stereotype Property extends the metaclass Property with multiplicity [1..1].

NOTE – A property is an attribute that corresponds to variables and agent instance sets in SDL, or fields of a structure.

7.12.1 Attributes

Stereotype attributes:

– initialNumber: UnlimitedNatural [0..1] defines the initial number of instances created when an instance of the containing classifier is created.

– referenceSort: Boolean determines the treatment of a variable or field as a value sort or reference sort and has a default value false.

7.12.2 Constraints

- The aggregation shall not be shared.
- If a <<Property>> Property has aggregation that is composite, the type shall be a <<PassiveClass>> Class with at least one ownedAttribute or an <<ActiveClass>> Class.
- The type shall not be omitted.
- If the upperValue is omitted, the lowerValue shall also be omitted.
- If the upperValue is included, the lowerValue shall also be included.

  NOTE – The upper and lower bounds on multiplicity are optional in UML-SS.
- If the type is an <<ActiveClass>> Class, the lowerValue shall be omitted or shall be zero.
- If the upperValue value is greater than 1 and isOrdered is true, isUnique shall be false, because there is not a predefined SDL data type that is ordered and requires each of its elements to have unique values.
- The initialNumber shall be included only if the type is an <<ActiveClass>> Class. The value of the InitialNumber shall not be greater than the upperValue.
- isDerived shall be false.
- isDerivedUnion shall be false.
- If isReadOnly is true, the type shall be a DataType or <<PassiveClass>> Class.
- The defaultValue shall be a constant expression.

7.12.3 Semantics

If isreadOnly is false and has an aggregationKind that is none and type is a <<PassiveClass>> Class or an Interface or a DataType (which includes PrimitiveType and Enumeration), the <<Property>> Property is mapped to a *Variable-definition*. The name defines the *Variable-name*. The defaultValue defines the *Constant-expression*. The *Sort-reference-identifier* is the *Sort-identifier* of the sort derived from the type property. The *Sort-identifier* is determined as follows:

– If there is no upperValue and no lowerValue, the name of the type maps to the *Sort-identifier*;

– Otherwise, the *Sort-identifier* identifies an anonymous sort formed from the SDL predefined `Bag` (if isOrdered is false and isUnique is false) or `Powerset` (if isOrdered is false and isUnique is true) or `String` (if isOrdered is true) datatype instantiated with the sort given by the type as the `ItemSort`. The anonymous sort is a *Value-data-type-definition* or *Syntype-definition* in the same context as the *Variable-definition*. If the upperValue value is omitted or the lowerValue value is zero and the upperValue value is unlimited (* in the concrete syntax), there are no size constraints and the anonymous sort is a *Value-data-type-definition* with its components derived from the instantiated predefined data type. Otherwise the lowerValue value and upperValue value map to a *Range-condition* of the anonymous sort, which is a *Syntype-definition*. The *Parent-sort-identifier* of this *Syntype-*

definition is a reference to another anonymous sort that is the *Value-data-type-definition* derived in the same way as the case with no size constraints.

If isreadOnly is true, the type is required to be either a DataType (which includes PrimitiveType and Enumeration) or a <<PassiveClass>> Class. When isreadOnly is true, the <<Property>> Property is mapped to a *Constant-expression* each time the <<Property>> Property is used in an expression. The defaultValue defines the *Constant-expression*.

If the type is an <<ActiveClass>> Class, the <<Property>> Property is mapped to an *Agent-definition*. The name defines the *Agent-name*. The type property defines the *Agent-type-identifier* that represents the type in the SDL abstract syntax. The initialNumber defines the *Initial-number*. The upperValue defines the *Maximum-number*. If the initialNumber is omitted, the lowerValue defines the *Initial-number*. If both the initialNumber and lowerValue are omitted, the *Initial-number* is 1.

NOTE 1 – It is possible for the number of instances to go below the *Initial-number*.

NOTE 2 – In UML the multiplicity of a property is separate from the type of the property; whereas in SDL, the bounds, uniqueness of values and ordering of elements are considered to be part of a data type and, if these differ, two types are considered to be different and incompatible. If two properties have the same type but have different bounds and both map to `Bags`, `Powersets` or `Strings`, the bounds are treated as a size constraints, so in these special cases two types could be compatible if they both had the same kind and item sort. The mappings defined above result in anonymous data types for each property, which has multiple values, with the consequence that such properties cannot be compatible even for the special cases. In SDL it is possible to define a type that has a specific name and item sort (and in the case of a Vector the upper bound) and to use this for different variable definitions so that the value of one variable can be assigned to another using the same type.

7.12.4 Notation

UML standard syntax is used with the following extensions. The property type <prop type> shall not be omitted. In a part symbol, the initialNumber is optionally specified as a slash followed by an <integer name> after the multiplicity if (and only if) the <prop type> denotes an <<ActiveClass>> Class.

<property> ::=
    [<visibility>] [<solidus>]
    <name> <colon> <prop type> [<multiplicity>]
    [<solidus> <integer name>]
    [<equals sign> <default>]

<multiplicity> ::=
    <left square bracket> <range condition> <right square bracket>
    [<left curly bracket>
        <order designator> [<comma> <uniqueness designator>]
    | <uniqueness designator> [<comma> <order designator>]
    <right curly bracket>]

<order designator> ::=
    ordered | **unordered**

<uniqueness designator> ::=
    unique | **nonunique**

<range condition> ::=
    <range> { <comma> <range> }*

<range> ::=
    <closed range>
  | <open range>
  | <asterisk>

<closed range> ::=
    <constant> <range sign> [<constant> | <asterisk>]

```
<open range> ::=
                <constant>
        |       {       <equality sign>
        |               <not equals sign>
        |               <less than sign>
        |               <greater than sign>
        |               <less than or equals sign>
        |               <greater than or equals sign> } <constant>
```

An <open range> that is <constant> is a shorthand form for <equality sign> <constant>.

An <asterisk> for a <range> or within an <open range> is valid only for a <range> for an UnlimitedNatural in a <multiplicity>. The <asterisk> represents an unlimited natural number.

```
<constant> ::=
                <expression>
```

7.12.5 References

SDL:	9	Agents
	12.3.1	Variable definition
	D.3.3	String
	D.3.9	Vector
	D.3.10	Powerset
	D.3.13	Bag
UML-SS:	7.3.32	MultiplicityElement (from Kernel)
	7.3.44	Property (from Kernel, Association Classes)
	7.3.49	StructuralFeature (from Kernel)
	7.3.52	TypedElement (from Kernel)

7.13 Signal

The stereotype Signal extends the metaclass Signal with multiplicity [1..1].

NOTE – A signal represents the type for communication message instances and maps to a *Signal-definition*.

7.13.1 Attributes

No additional attributes.

7.13.2 Constraints

- A <<Signal>> Signal shall not have operations.

7.13.3 Semantics

A <<Signal>> Signal is mapped to a *Signal-definition*. The Name defines the *Signal-name*. The type of each ownedAttribute defines the corresponding *Sort-reference-identifier*.

7.13.4 Notation

UML standard syntax is used.

7.13.5 References

SDL:	10.3	Signal
UML-SS:	13.3.24	Signal (from Communications)

7.14 Timer

The Timer stereotype is a subtype of the stereotype Signal.

7.14.1 Attributes

Stereotype attributes:

– default: Duration This represents the default duration for the timer.

7.14.2 Constraints

No additional constraints.

7.14.3 Semantics

A <<Timer>> Signal maps to a *Timer-definition*. The name attribute defines the *Timer-name*. The type of each ownedAttribute defines the corresponding *Sort-reference-identifiers*.

7.14.4 Notation

The notation for a timer is a classifier symbol with the keyword <<timer>>.

7.14.5 References

SDL: 11.15 Timer

8 State machines

The finite state machine models of SDL-UML provide details of how a model behaves in terms of state transitions for the protocol part of a system.

The following metaclasses from the UML package BehaviorStateMachines are included:

– FinalState

– Pseudostate

– Region

– State

– StateMachine

– Transition

8.1 FinalState

The stereotype FinalState extends the metaclass FinalState with multiplicity [1..1].

NOTE – When a FinalState is reached the containing graph completes. In SDL-UML a graph for a procedure will complete with a <<Return>> ActivityFinalNode. In this case, there is no mapping to the SDL abstract syntax for FinalState, because the return node terminates the graph. A FinalState that is not in a procedure graph maps to an *Action-return-node* or *Named-return-node* for the enclosing composite state.

8.1.1 Attributes

No additional attributes.

8.1.2 Constraints

• If the <<FinalState>> FinalState is part of the region of a <<StateMachine>> StateMachine that maps to a *Procedure-graph*, the name of the <<FinalState>> FinalState shall be empty and any Transition that has the <<FinalState>> FinalState as its target shall end in a <<Return>> ActivityFinalNode.

NOTE – The *Action-return-node* or *Value-return-node* of the procedure is defined by the <<Return>> ActivityFinalNode.

8.1.3 Semantics

If the <<FinalState>> FinalState has an empty name and it is not part of the region of a <<StateMachine>> StateMachine that maps to a *Procedure-graph*, the <<FinalState>> FinalState is mapped to a *Stop-node* or an *Action-return-node*. It is mapped to a *Stop-node* if (and only if) it is part of the region of a <<StateMachine>> StateMachine that is the classifierBehavior of an <<ActiveClass>> Class.

NOTE – In UML FinalState the context object of the state machine is terminated if all enclosed regions are terminated, whereas in SDL an explicit stop is required, but, on the other hand, in SDL it is not allowed to have a return node in the state machine of an agent.

If the <<FinalState>> FinalState has a non-empty name, it is mapped to a *Named-return-node* where the name defines the *State-exit-point-name*.

8.1.4 Notation

UML standard syntax is used.

8.1.5 References

SDL: 11.12.2.4 Return

UML-SS: 15.3.2 FinalState (from BehaviorStateMachines)

8.2 Pseudostate

The stereotype Pseudostate extends the metaclass Pseudostate with multiplicity [1..1].

NOTE – A Pseudostate is used instead of a state before initial or state entry point transitions, when there is a junction of transitions, when there is a decision to make a choice of transitions, when the transition leads to a history nextstate, or after a transition to lead to a state exit point or terminate the state graph. They allow more complex transitions between states to be built from simpler, shorter transitions that end or start (or start and end) in a Pseudostate. They map to start, next state (with history), decision, join and free action, return and stop nodes in the SDL state transition graph.

8.2.1 Attributes

No additional attributes.

8.2.2 Constraints

- A Transition shall have an empty guard property if the Transition is an outgoing property of a <<Pseudostate>> Pseudostate with kind == initial.
- A Transition shall have an empty trigger property if the Transition is an outgoing property of a <<Pseudostate>> Pseudostate with kind == initial.
- The classifierBehavior of a non-abstract <<ActiveClass>> Class shall have a <<Pseudostate>> Pseudostate with kind == initial.
- The kind property of <<Pseudostate>> Pseudostate shall not be join or fork or shallowHistory.
- A <<Pseudostate>> Pseudostate with kind == deepHistory or with kind == exitPoint or with kind == terminate shall not have an outgoing property.
- A Transition shall have a non-empty guard property Constraint (a Boolean Expression) if the Transition is an outgoing property of a <<Pseudostate>> Pseudostate with kind == choice.
- Each guard of each Transition that is an outgoing property of a <<Pseudostate>> Pseudostate with kind == choice shall be an Expression with two operand properties. One operand shall be identical in every such guard of the <<Pseudostate>> Pseudostate, and for the purposes of description is called the left-hand operand. For the purposes of description

the other operand is called the right-hand operand, and shall evaluate to a value set (possibly with just one element) with elements of the same data type as the left-hand operand. The value set defined by a right-hand operand shall be statically determinable.

8.2.3 Semantics

A <<Pseudostate>> Pseudostate with kind == initial is mapped to a *Procedure-start-node* in a region that defines a *Procedure-graph* and *State-start-node* in a region that defines a *Composite-state-graph*. The outgoing property maps to the *Graph-node* list of the *Transition* of the *Procedure-start-node* or *State-start-node*. The target property of this outgoing property Transition maps to the last item of the *Transition* (a *Terminator* or *Decision-node*) of the *Procedure-start-node* or *State-start-node* in the same way as the target is mapped in clause 8.6 for a Transition.

A <<Pseudostate>> Pseudostate with kind == deepHistory is mapped to a *Nextstate-node* that is a *Dash-nextstate* with **HISTORY**.

A <<Pseudostate>> Pseudostate with kind == junction is mapped to a *Free-action* and one or more *Join-node* elements. The name property defines the *Connector-name* in the *Free-action* and each *Join-node*. The effect of the outgoing property maps to the *Graph-node* list of the *Transition* of the *Free-action*. The target property of this outgoing property Transition maps to the last item of the *Transition* (a *Terminator* or *Decision-node*) of the *Free-action* in the same way as the target is mapped in clause 8.6 for a Transition. There is a *Join-node* for each Transition that has a target property that is a <<Pseudostate>> Pseudostate with kind == junction and the *Join-node* is the *Terminator* of the *Transition* with its *Graph-node* list derived from the effect of the Transition.

NOTE – UML-SS has a constraint "a junction vertex must have at least one incoming and one outgoing transition". Pseudostate maps to both the *Join-node* elements and the *Free-action* labels, so the possibility (allowed in [ITU-T Z.100]) to have a *Free-action* without a corresponding *Join-node* is not allowed.

A <<Pseudostate>> Pseudostate with kind == choice is mapped to a *Decision-node*. The outgoing property Transition maps to the *Decision-question* and *Decision-answer-**set***. The common left-hand operand (see Constraints above) of the guard properties of the outgoing properties maps to the *Decision-question*. The right-hand operand (see Constraints above) of a guard property of an outgoing property Transition maps to the *Range-condition* of a *Decision-answer* and the effect of this outgoing property maps to the *Graph-node* list of the *Transition* of the same *Decision-answer*. The target property of this outgoing property Transition maps to the last item of the *Transition* (a *Terminator* or *Decision-node*) of the same *Decision-answer* in the same way as the target is mapped in clause 8.6 for a Transition.

A <<Pseudostate>> Pseudostate with kind == entryPoint is mapped to a *Start-state-node*. The name property defines the *State-entry-point-name*. The effect of the outgoing property defines the *Graph-node* list of the *Transition*. The target property of this outgoing property Transition maps to the last item of the *Transition* (a *Terminator* or *Decision-node*) of the *Start-state-node* in the same way as the target is mapped in clause 8.6 for a Transition.

A <<Pseudostate>> Pseudostate with kind == exitPoint is mapped to a *Named-return-node*. The name property defines the *State-exit-point-name*.

A <<Pseudostate>> Pseudostate with kind == terminate is mapped to a *Stop-node*.

8.2.4 Notation

UML standard syntax is used. The notation for a <<Pseudostate>> Pseudostate with kind == choice is as shown in the example in UML-SS Figure 15.23. In this example, "id" corresponds to the left-hand operand described in the Constraints clause above and ">=10" and "<10" to the right-hand operand. The form on the left of UML-SS Figure 15.23 is preferred. The left-hand operand can be an expression of any complexity. The expression for a right-hand operand is typically a single value or defines a range of values. In this example, "id" is a question and ">=10" and "<10" are answers:

the syntax for <question> and <answer> of clause 9.13.4 <<For>> LoopNode shall be used in these contexts.

NOTE – Neither UML shallow history nor SDL dash next state notations are supported, so that to return to the current state, the state name must be given explicitly.

8.2.5 References

SDL:	11.1	Start
	11.12.2.2	Join
	11.10	Label
	11.13.15	Decision
	11.12.2.3	Stop
UML-SS:	15.3.8	Pseudostate (from BehaviorStateMachines)
	15.3.9	PseudostateKind (from BehaviorStateMachines)

8.3 Region

The stereotype Region extends the metaclass Region with multiplicity [1..1].

NOTE – A region contains states and transitions and is mapped to the definition of how a procedure or a composite state behaves. For the composite state mapping of a StateMachine, a single region maps to a *Composite-state-graph*, whereas two or more regions map to a *State-aggregation-node* (see clause 8.5). A region in SDL-UML is always part of a StateMachine and is never part of a State, because the region of a State is constrained to be empty.

8.3.1 Attributes

No additional attributes.

8.3.2 Constraints

- A Region that extends another Region (as specified by an extendedRegion property) shall have the same name as the extended Region.
- The triggers in the different orthogonal regions shall refer to disjoint sets of signals.

8.3.3 Semantics

A <<Region>> Region that is the region of StateMachine with a specification is mapped to a *Procedure-graph*, and the subvertex set of Vertex elements (State, Pseudostate, or FinalState) of the region together with the transition elements of the region that reference these Vertex elements define the *Procedure-graph*.

A <<Region>> Region that is the only region of a StateMachine without a specification is mapped to a *Composite-state-graph*, and the subvertex set of Vertex elements (State, Pseudostate, or FinalState) of the region together with the transition elements of the region that reference these Vertex elements define the *Composite-state-graph* of the StateMachine mapping. Each *State-node* or *Free-action* derived from these Vertex elements are elements of the *State-node-**set*** and *Free-node-**set***, respectively of the *State-transition-graph* of the *Composite-state-graph*.

Otherwise, each <<Region>> Region that is one of two or more regions of a StateMachine without a specification is mapped to a *State-partition* and to a *Composite-state-type-definition* with a unique *State-type-name*. Each *State-partition* is an element of the *State-partition-**set*** of the *State-aggregation-node* of the *Composite-state-type-definition* of the StateMachine mapping. The mapping to a *State-partition* and the corresponding inner *Composite-state-type-definition* is described in more detail in the following paragraphs.

Each Pseudostate with kind entryPoint (in the connectionPoint property of the containing StateMachine) maps to a distinct *State-entry-point-definition* of the *Composite-state-type-definition*. The *Connection-definition-**set*** of the *State-partition* contains an *Entry-connection-definition* that connects the *State-entry-point-definition* of the outer *Composite-state-type-definition* to the corresponding *State-entry-point-definition* of the inner *Composite-state-type-definition*.

Each Pseudostate with kind exitPoint in the connectionPoint property of the containing StateMachine maps to a distinct *State-exit-point-definition* of the *Composite-state-type-definition*. The *Connection-definition-**set*** of the *State-partition* contains an *Exit-connection-definition* that connects the *State-exit-point-definition* of the outer *Composite-state-type-definition* to the corresponding *State-exit-point-definition* of the inner *Composite-state-type-definition*.

The Name maps to the *Name* of the *State-partition*.

The *Composite-state-type-identifier* of the *State-partition* identifies the inner *Composite-state-type-definition*.

The subvertex and transition properties of the Region map to the *Composite-state-graph* of the inner *Composite-state-type-definition* in the same way that a *Composite-state-graph* is derived for only one region in a StateMachine. See the clauses covering subclasses of Vertex (that is State, Pseudostate, or FinalState) and the Transition clause for more details.

8.3.4 Notation

UML standard syntax is used.

8.3.5 References

SDL:	8.1.1.5	Composite state type
	11.11.2	State aggregation
UML-SS:	13.3.2	Behavior (from BasicBehaviors)
	15.3.10	Region (from BehaviorStateMachines)

8.4 State

The stereotype State extends the metaclass State with multiplicity [1..1].

NOTE – A state represents a condition where an object is waiting for some condition to be fulfilled: usually for an event to occur. A state in SDL-UML maps to an SDL state.

8.4.1 Attributes

No additional attributes.

8.4.2 Constraints

- The doActivity property shall be empty.
- The entry and exit properties shall be empty, because entry/exit actions are not supported.
- The isComposite property shall be false, because only decomposition using submachine properties is allowed and a State shall have an empty region property.
- In the Transition set defined by the outgoing properties of a State, the signal property of each event property that is a SignalEvent of each trigger shall be distinct.

8.4.3 Semantics

A <<State>> State is mapped to a *State-node*.

The name maps to the *State-name*.

A ConnectionPointReference that is part of the connection property and corresponds to an *Exit-Connection-Point* (a Pseudostate with kind exitPoint in the connectionPoint property of the containing StateMachine) maps to a member of the *Connect-node-**set***.

The submachine property maps to *Composite-state-type-identifier*.

A deferrableTrigger property maps to an element of the *Save-signal-**set***.

The outgoing property (inherited from Vertex) maps to the *Input-node-**set***, *Spontaneous-transition-**set*** and *Continuous-signal-**set***. See clause 8.6 on Transition for more details on the mapping to the *Input-node-**set***, *Spontaneous-transition-**set*** and *Continuous-signal-**set***.

8.4.4 Notation

UML standard syntax is used.

8.4.5 References

SDL: 11.2 State

UML-SS: 15.3.11 State (from BehaviorStateMachines, ProtocolStateMachines)

15.3.16 Vertex (from BehaviorStateMachines)

8.5 StateMachine

The stereotype StateMachine extends the metaclass StateMachine with multiplicity [1..1].

NOTE – An SDL-UML StateMachine either maps to the graph of an SDL procedure or an SDL composite state type. The two cases are distinguished by whether or not the StateMachine has a specification. If it does, then it is the procedure case; otherwise, it is a composite state type. Because there are two different mappings, some constraints on StateMachine are dependent on whether there is a specification or not.

8.5.1 Attributes

No additional attributes.

8.5.2 Constraints

- Each ownedAttribute property shall have an aggregation that is composite.

NOTE 1 – As a consequence, the part properties are the same as the ownedAttribute properties.

- The isReentrant property shall be false.
- If the StateMachine has a specification property, the specification property shall be an Operation.

NOTE 2 – The other possibility, Reception, is not allowed.

- If the StateMachine has a specification property, the ownedParameter list of the StateMachine shall be the same as the ownedParameter list of the Operation that is the specification property.
- The ownedConnector shall be empty.
- The redefinedClassifier property shall be empty.
- If the StateMachine redefines another Behavior (as specified by redefinedBehavior), the Behavior shall be a StateMachine.
- If the StateMachine redefines another StateMachine (as specified by redefinedBehavior, or extendedStateMachine), it shall have the same name as the redefined StateMachine.
- If the StateMachine is the classifierBehavior of a Class, the redefinedBehavior property shall be empty.
- If the StateMachine is not the classifierBehavior of a Class, then the extendedStateMachine property shall be empty.

If a StateMachine is mapped to a *Composite-state-type* (see the Semantics clause below):

- The returnedResult property shall be empty (so that StateMachine does not return a result).

If a StateMachine is mapped to a *Procedure-graph* (see the Semantics clause below):

- There shall only be one Region.
- The connectionPoint property shall be empty.
- The classifierBehavior shall be empty.
- The ownedPort shall be empty.
- The general property shall be empty.

NOTE 3 – A *Procedure-graph* never inherits directly from another graph. Instead, a *Procedure-definition* mapped from an <<Operation>> Operation with a general property that is not empty has a *Procedure-identifier* for an inherited *Procedure-definition* mapped from the general property, and inherits the *Procedure-graph* of this *Procedure-definition*.

- The specification shall not be an Operation contained in an Interface.

8.5.3 Semantics

A <<StateMachine>> StateMachine is mapped to a *Composite-state-type-definition* or a *Procedure-graph*. If the StateMachine has a specification, the StateMachine is mapped to the *Procedure-graph* (as defined by its contained Region) of the *Procedure-definition* from the mapping of the <<Operation>> Operation identified by the specification. If the StateMachine does not have a specification, the StateMachine is mapped to a *Composite-state-type-definition*.

Semantics for the *Procedure-graph* case (where the *Procedure-definition* is the mapping of <<Operation>> Operation identified by the specification):

The region property defines the *Procedure-graph* through the subvertex set of Vertex elements (State, Pseudostate, or FinalState) of the region together with the transition elements of the region that reference these Vertex elements. Each *State-node* or *Free-action* derived from these Vertex elements are elements of the *State-node-**set*** and *Free-node-**set*** respectively of the *Procedure-graph*.

NOTE 1 – A Pseudostate with kind initial defines the *Procedure-start-node*.

The nestedClassifier and ownedAttribute associations (both inherited from Class via Behavior) define the rest of the contents of the state machine according to the following paragraphs.

A nestedClassifier that is a Datatype, EnumerationType or PrimitiveType defines a *Value-data-type-definition* that is an element of the *Data-type-definition-**set*** of the *Procedure-definition*.

A nestedClassifier that is an Interface defines an *Interface-definition* that is an element of the *Data-type-definition-**set*** of the *Procedure-definition*.

A nestedClassifier that is a <<PassiveClass>> Class defines an *Object-type-definition* that is an element of the *Data-type-definition-**set*** of the *Procedure-definition*.

A nestedClassifier that is a <<StateMachine>> StateMachine defines a *Composite-state-type-definition* that is an element of the *Composite-state-type-definition-**set*** of the *Procedure-definition*.

An ownedOperation defines a *Procedure-definition* that is an element of the *Procedure-definition-**set*** of the *Procedure-definition* mapping the Operation identified by the specification.

An ownedAttribute maps to a *Variable-definition* in the *Variable-definition-**set*** of the *Procedure-definition*.

Semantics for the *Composite-state-type-definition* case:

The name defines the *State-type-name*. If the region contains only one Region, the content of the region is mapped to a *Composite-state-graph* of the *Composite-state-type-definition*, otherwise the region maps to a *State-aggregation-node* of the *Composite-state-type-definition* with one *State-partition* for each contained Region.

Each connectionPoint with kind entryPoint defines an element of the *State-entry-point-definition-**set*** and each connectionPoint with kind exitPoint defines an element of *State-exit-point-definition-**set***.

The ownedParameter property defines the *Composite-state-formal-parameter*s.

The nestedClassifier and ownedAttribute associations define the rest of the contents of the state machine according to the following paragraphs.

A nestedClassifier that is a Datatype, EnumerationType or PrimitiveType defines a *Value-data-type-definition* that is an element of the *Data-type-definition-**set***.

A nestedClassifier that is an Interface defines an *Interface-definition* that is an element of the *Data-type-definition-**set***.

A nestedClassifier that is a <<PassiveClass>> Class defines an *Object-type-definition* that is an element of the *Data-type-definition-**set***.

A nestedClassifier that is a <<StateMachine>> StateMachine defines a *Composite-state-type-definition* that is an element of the *Composite-state-type-definition-**set***.

An ownedOperation defines a *Procedure-definition* that is an element of the *Procedure-definition-**set***.

An ownedAttribute maps to a *Variable-definition* in the *Variable-definition-**set***.

The general property (derived from generalization) maps to the optional *Composite-state-type-identifier*.

NOTE 2 – If a StateMachine is a classifierBehavior and it has an ownedParameter set, these parameters are used as parameters when creating instances of the containing Class. See clause 7.1.3 the semantics for ActiveClass.

8.5.4 Notation

UML standard syntax is used.

8.5.5 References

SDL:	8.1.1.5	Composite state type
	9.5	Procedure
UML-SS:	13.3.2	Behavior (from BasicBehaviors)
	13.3.4	BehavioredClassifier (from BasicBehaviors, Communications)
	15.3.12	StateMachine (from BehaviorStateMachines)

8.6 Transition

The stereotype Transition extends the metaclass Transition with multiplicity [1..1].

NOTE – A transition is the part of a state transition graph that defines what happens when the object goes from one vertex in the graph to another vertex. Each vertex is usually a state, but may be a pseudostate. Signals (including timer signals) timers are used to trigger transitions. Standard UML notation and semantics are used.

8.6.1 Attributes

Stereotype attributes:

– priority: UnlimitedNatural

8.6.2 Constraints

– The Transition shall have kind == external or local. The UML concept of internal transitions is not allowed.

– The trigger property shall not be empty.

– The port of the Trigger that is the trigger property of the Transition shall be empty.

– The event property of the trigger property shall be a MessageEvent or ChangeEvent.

– The effect property shall reference an Activity.

NOTE – There is a constraint on states that signals for each transition have to be distinct, so that a given signal is not allowed to trigger more than one transition.

8.6.3 Semantics

In this clause the term 'trigger event of a <<Transition>> Transition' means the Event that is the event property of the Trigger that is the trigger property of the Transition. The Event is a MessageEvent (an AnyReceiveEvent, a SignalEvent, or a CallEvent) or ChangeEvent.

If the <<Transition>> Transition has a Transitionkind that is local, it is expanded according to the mapping rules given for asterisk state list in the Model clause in 11.2 in [ITU-T Z.100] before applying any expansions or mappings below.

If the trigger event of a <<Transition>> Transition is an AnyReceiveEvent, the transition is expanded according to the Model in SDL 11.3 (for transforming asterisk input list) before applying any expansions or mappings below.

If the trigger event of a <<Transition>> Transition is a CallEvent, the transition is expanded according to the Model in SDL 10.5 before any expansions or mappings below.

If the trigger event of a <<Transition>> Transition is a SignalEvent and the name of the Signal is "none" or "NONE" (case sensitive therefore excludes "None"), the Transition is mapped to a *Spontaneous-transition*. The effect property maps to the *Graph-node* list of the *Transition* of the *Spontaneous-transition*.

If the trigger event of a <<Transition>> Transition is a SignalEvent and the name of the Signal is neither "none" nor "NONE" (so it does not map to *Spontaneous-transition*), the Transition is mapped to an *Input-node*. The qualifiedName of the Signal maps to the *Signal-identifier* of the *Input-node*, and for each <attr name> in the <assignment specification> (see the Notation given in UML-SS 13.3.25) the qualifiedName of the attribute (with this name) of the context object owning the triggered behavior is mapped to the corresponding (by order) *Variable-identifier* of the *Input-node*. The effect property maps to the *Graph-node* list of the *Transition* of the *Input-node*.

NOTE 1 – There is no UML meta-model element that corresponds simply and directly to the <attr name>. Instead the UML-SS informally relates the syntax element to the attribute of the context object.

If the trigger event of a <<Transition>> Transition is a ChangeEvent, the transition is mapped to a *Continuous-signal*. The changeExpression maps to the *Continuous-expression* of the *Continuous-signal*. The effect property maps to the *Graph-node* list of the *Transition* of the *Continuous-signal*. The priority maps to the *Priority-name*.

If the <<Transition>> Transition has an empty trigger property and a non-empty guard property, the Transition is mapped to a *Continuous-signal*. The guard maps to the *Continuous-expression* of the *Continuous-signal*. The effect property maps to the *Graph-node* list of the *Transition* of the *Continuous-signal*. The priority maps to the *Priority-name*.

NOTE 2 – It is a consequence of the SDL semantics that in the Transition set defined by the outgoing properties of a State, when evaluating the guard of each *Continuous-signal* (each Transition with only a guard and an empty trigger), an unevaluated guard of a Transition with a lowest priority attribute is evaluated before any unevaluated guard of a Transition with a higher priority attribute.

If the <<Transition>> Transition has an empty trigger property and an empty guard property, the Transition is mapped to a *Connect-node*. The effect property maps to the *Graph-node* list of the *Transition* of the *Connect-node*. If the source of the Transition is a ConnectionPointReference, the qualifiedName of the exit property Pseudostate of the ConnectionPointReference maps to *State-exit-point-name*. If the source is a State, the *State-exit-point-name* is empty.

If a <<Transition>> Transition has a non-empty trigger property and non-empty guard property, the guard is mapped to the *Transition* as follows. A *Decision-node* is inserted first in the *Transition* with a *Decision-answer* with a Boolean *Range-condition* that is the *Constant-expression* true and another *Decision-answer* for false. The specification property of the guard property of the Transition maps to *Decision-question* of the *Decision-node*. The false *Decision-answer* has a *Transition* that is a *Dash-nextstate* without **HISTORY**. The effect property of the Transition maps to the *Graph-node* list of the *Transition* of the true *Decision-answer*.

NOTE 3 – The mapping to a *Decision-node* instead of mapping to an enabling condition (a *Provided-expression*) makes it possible to access the signal parameters from the expression in the guard and also means that the signal is consumed even if guard is false, whereas if an enabling condition is false the signal is not consumed.

NOTE 4 – The mapping to a *Decision-node* works because entry/exit actions are not allowed on states. If such actions were allowed, the exit and entry actions of the states would be incorrectly invoked even when taking the false branch through the decision.

A target property that is a State maps to a *Terminator* of the *Transition* (mapped from the effect) where this *Terminator* is a *Nextstate-node* that is a *Named-nextstate* without *Nextstate-parameters,* and where the qualifiedName of the State maps to the *State-name* of the *Named-nextstate*.

A target property that is a ConnectionPointReference maps to a *Terminator* of the *Transition* (mapped from the effect) where this *Terminator* is a *Nextstate-node* that is a *Named-nextstate* with *Nextstate-parameters,* and where the qualifiedName of the state property of the ConnectionPointReference maps to the *State-name* of the *Named-nextstate*, and the qualifiedName of the entry property Pseudostate of the ConnectionPointReference maps to *State-entry-point-name* of the *Nextstate-parameters*.

A target property that is a Pseudostate maps to the last item of the *Transition* (a *Terminator* or *Decision-node*) as defined in clause 8.2, Pseudostate.

8.6.4 Notation

UML standard syntax is used.

8.6.5 References

SDL: 11.3 Input

11.9 Spontaneous transition

11.5 Continuous signal

UML-SS: 13.3.25 SignalEvent (from Communications)

13.3.31 Trigger (from Communications)

15.3.1 ConnectionPointReference (from BehaviorStateMachines)

15.3.14 Transition (from BehaviorStateMachines)

9 Actions and activities

An activity is used to describe how the model behaves, for example the control flow of actions in an operation body or a transition. When invoked, each action takes zero or more inputs, usually modifies the state of the system in some way such as a change of the values of an instance, and produces zero or more outputs. The values that are used by an action are described by value specifications (see clause 10, ValueSpecification), obtained from the output of actions or in ways specific to the action. The UML specification contains a framework for dealing with actions, but does not provide syntax. In the stereotypes below, the syntax is given for actions, and these actions are mapped to the UML framework.

The following packages from UML are included either explicitly or because elements of the packages are generalizations that are specialized as the elements that are used:

– BasicActions
– BasicActivities
– BasicBehaviors
– CompleteActivities
– CompleteStructuredActivities
– FundamentalActivities
– IntermediateActivities
– IntermediateActions
– StructuredActions
– StructuredActivities

The following metaclasses from UML are included:

– Activity
– ActivityFinalNode
– AddStructuralFeatureValueAction
– AddVariableValueAction
– CallOperationAction
– CreateObjectAction
– ConditionalNode
– LoopNode
– OpaqueAction
– SendSignalAction
– SequenceNode

9.1 Activity

The stereotype Activity extends the metaclass Activity with multiplicity [1..1].

NOTE – An activity defines the effect of a transition or the body of an operation.

9.1.1 Attributes

No additional attributes.

9.1.2 Constraints

• An <<Activity>> Activity shall be empty or contain at most one ActivityNode in its node property and this node shall be a SequenceNode.

9.1.3 Semantics

An <<Activity>> Activity that is the effect of a Transition is mapped to the *Graph-node* list of the *Transition* for the effect.

An <<Activity>> Activity that has a specification (that is, the Activity is the method of a BehavioralFeature) is mapped to a *Procedure-graph* containing only a *Procedure-start-node* consisting of a *Transition*.

The actions contained in the SequenceNode map to the *Graph-node* list of the *Transition*.

NOTE – See clause 7.7.3 for the mapping of operations to a *Procedure-definition* that have a method defined by an <<Activity>> Activity.

9.1.4 Notation

UML standard syntax is used.

9.1.5 References

SDL: 11.12 Transition

UML-SS: 12.3.4 Activity (from BasicActivities, CompleteActivities, FundamentalActivities, StructuredActivities)

9.2 ActivityFinalNode

The stereotype ActivityFinalNode extends the metaclass ActivityFinalNode with multiplicity [1..1]. This stereotype is abstract.

NOTE – This abstract stereotype is introduced to ensure that every ActivityFinalNode is one of the subtypes: <<Return>> ActivityFinalNode or <<Stop>> ActivityFinalNode. As the stereotype is abstract, each instance has to be one of the concrete subtypes.

9.2.1 Attributes

No additional attributes.

9.2.2 Constraints

No additional constraints.

9.2.3 Semantics

The concrete subtypes of the stereotype ActivityFinalNode give its semantics.

9.2.4 Notation

The concrete subtypes of the stereotype ActivityFinalNode give its notation.

9.2.5 References

UML-SS: 12.3.6 ActivityFinalNode (from BasicActivities, IntermediateActivities)

9.3 AddStructuralFeatureValueAction

The stereotype AddStructuralFeatureValueAction extends the metaclass AddStructuralFeatureValueAction with multiplicity [1..1].

NOTE – An <<AddStructuralFeatureValueAction>> AddStructuralFeatureValueAction is used to define an assignment to structural features of a Class or other Classifier.

9.3.1 Attributes

Stereotype attributes:

– assignmentAttempt: Boolean — If true, the AddStructuralFeatureValueAction represents an assignment attempt.

9.3.2 Constraints

- The value property shall be a ValuePin.
- The object property shall be a ValuePin.

9.3.3 Semantics

An <<AddStructuralFeatureValueAction>> AddStructuralFeatureValueAction is mapped to a *Task-node* that is an *Assignment* (if the assignmentAttempt property is false) or an *AssignmentAttempt* (if the assignmentAttempt property is true). The value property maps to the *Expression* of the *Assignment* or *AssignmentAttempt* (respectively). The qualifiedName of the structuralFeature property maps to the *Variable-identifier*.

The object property together with the structuralFeature property should be transformed according to the Model in clause 12.3.3.1 of [ITU-T Z.100] before mapping to the SDL abstract syntax. This is the situation where the <variable> is a <field primary> or <indexed primary>.

9.3.4 Notation

When an <<AddStructuralFeatureValueAction>> AddStructuralFeatureValueAction is defined in textual syntax (for example when used inside an action symbol), textual notation is used. The textual notation should follow the following grammar:

```
<structural feature assignment statement> ::=
                              <assignment>
                    |         <assignment attempt>

<assignment> ::=
                              <variable> <is assigned sign> <expression>

<assignment attempt> ::=
                              <variable> <is assigned sign> as <less than sign> <identifier> <greater than sign>
                                   <left parenthesis> <expression> <right parenthesis>

<variable> ::=
                              <identifier>
                    |         <field primary>
                    |         <indexed primary>
```

9.3.5 References

SDL: 12.3.3 Assignment and assignment attempt

UML-SS: 11.3.5 AddStructuralFeatureValueAction (from IntermediateActions)

11.3.47 StructuralFeatureAction (from IntermediateActions)

9.4 AddVariableValueAction

The stereotype AddVariableValueAction extends the metaclass AddVariableValueAction with multiplicity [1..1].

NOTE – AddVariableValueActions are used to define assignment to local variables of compound statements.

9.4.1 Attributes

Stereotype attributes:

– assignmentAttempt: Boolean — If true, the AddVariableValueAction represents an assignment attempt.

9.4.2 Constraints

- The InputPin in the value property shall be a ValuePin.

9.4.3 Semantics

A <<AddVariableValueAction>> AddVariableValueAction is mapped to an *Assignment* (if the assignmentAttempt property is false) or to an *AssignmentAttempt* (if the assignmentAttempt property is true). The value property defines the *Expression* (through the contained ValueSpecification). The variable property defines the *Variable-identifier*.

9.4.4 Notation

When an <<AddVariableValueAction>> AddVariableValueAction is defined in textual syntax (for example when used inside a task box), the textual notation should follow the following grammar:

```
<variable assignment statement> ::=
                <assignment>    |    <assignment attempt>
```

The left-hand side of the assignment or assignment attempt shall not be a <field primary> or <indexed primary>.

9.4.5 References

SDL: 12.3.3 Assignment and assignment attempt

UML-SS: 11.3.6 AddVariableValueAction

11.3.52 VariableAction

9.5 Break

The stereotype Break is a concrete subtype of the stereotype OpaqueAction.

NOTE – A <<Break>> OpaqueAction represents a break action within a loop that causes termination of the enclosing loop labelled by the name given.

9.5.1 Attributes

The <<Break>> OpaqueAction has the following attribute:

– label: String The name of the loop to break out of.

9.5.2 Constraints

- A <<Break>> OpaqueAction shall have an empty input property.
- A <<Break>> OpaqueAction shall only exist inside the bodyPart of a LoopNode that has a name with a value equal to the label.

9.5.3 Semantics

A <<Break>> OpaqueAction is mapped to a *Break-node*. The *Connector-name* is the *Connector-name* of the containing LoopNode that has a name with a value equal to the label. See also clauses 9.13 and 9.21.

9.5.4 Notation

When a <<Break>> OpaqueAction is defined in textual syntax (for example when nested inside a task box), the following textual notation is used:

```
<break statement> ::=
                break [ <name> ] <semicolon>
```

The label is defined by the <name> part of the textual syntax, if present. If there is no <name>, the label has the same value as the name of the directly enclosing LoopNode.

9.5.5 References

SDL: 11.14.1 Compound statement

11.14.6 Loop statement

9.6 CallOperationAction

The stereotype CallOperationAction extends the metaclass CallOperationAction with multiplicity [1..1].

NOTE – A call operation action maps to the call of a procedure in the SDL abstract grammar.

For the description in this clause, the following terminology is used:

- The operation-owner is the Class that has (as an ownedOperation property) the Operation identified by the operation property of the <<CallOperationAction>> CallOperationAction.
- The active-container is the closest containing <<Active>> Class of the CallOperationAction.

9.6.1 Attributes

No additional attributes.

9.6.2 Constraints

- The target property shall be a ValuePin.
- If the CallOperationAction maps to a *Call-node*, the target property shall have the same Class as the operation-owner, because in this case the CallOperationAction represents the invocation of a method that acts on the item identified by the target property. For such a method invocation, if the target property has an <<Active>> Class, the actual target shall be an InstanceValue that identifies the <<Active>> Class instance.
- The onPort attribute shall be absent if the CallOperationAction maps to a *Call-node*.
- If the CallOperationAction does not map to a *Call-node*, the target property shall have an <<Active>> Class and the actual target shall be an InstanceValue that identifies an <<Active>> Class instance.

9.6.3 Semantics

A <<CallOperationAction>> CallOperationAction is mapped to a *Call-node* if:

- The operation-owner is a <<PassiveClass>> Class or a DataType, or
- The active-container is the same as the operation-owner or is a generalization of the operation-owner.

For mapping to a *Call-node*, the qualifiedName of the operation property is mapped to the *Procedure-identifier* of the *Call-node*, the target property is mapped to the first item of the *Expression* list of the *Call-node*, and the argument properties map to the *Expression* list (if the target property is absent) or the remainder of the *Expression* list (if the target property is present).

If the criteria for mapping to a *Call-node* are not satisfied and the <<CallOperationAction>> CallOperationAction is not invoked as part of an expression, the <<CallOperationAction>> CallOperationAction is transformed to a signal exchange, so that the nodes below replace the <<CallOperationAction>> CallOperationAction. This corresponds to the Model in clause 10.5 of [ITU-T Z.100] for a remote procedure call.

If the criteria for mapping to a *Call-node* are not satisfied and the <<CallOperationAction>> CallOperationAction returns a value and is invoked as part of an expression, the <<CallOperationAction>> CallOperationAction is transformed to a call of an implicitly defined local operation that has the nodes described below as its body and returns the value of the implicit variable that received the value from `pREPLY`. This corresponds to the Model in clause 12.3.5 of [ITU-T Z.100] for a value returning procedure call that contains a remote procedure call body.

The following nodes are used as the body of the implicit operation where each node is the outgoing node of the preceding node (except for the false branch of <<Pseudostate>> Pseudostate with kind == choice):

- An action `n := n+1`, where `n` is an implicit integer variable attribute of the active-container. The variable `n` is initialized to 0 and is used to recognize and discard replies from previous operation calls.
- A SendSignalAction to send an implicit signal `pCALL` for invoking the operation, where `p` is uniquely determined for this operation and this signal and a corresponding signal for the reply, `pREPLY`, are defined in a scope such as the signals are visible to both the sender and receiver. The `pCALL` signal has the same parameters as the original operation omitting parameters corresponding to **out** parameters and an additional last parameter that is an Integer. The signal is sent as:

  ```
  pCALL(apar,n)
  ```

  where `apar` is the original actual parameter list of the operation call omitting parameters corresponding to **out** parameters.

  The target property and onPort property of the SendSignalAction are the same as the corresponding property of the <<CallOperationAction>> CallOperationAction. The qualifiedName of signal property of the SendSignalAction identifies `pCALL`.
- An implicit State with an anonymous name with a set of deferrableTrigger properties that includes all signals that can be received except the signal `pREPLY`.
- A Transition that is a SignalEvent for the Signal `pREPLY`. The `pREPLY` signal has formal and actual parameters (`aINOUTpar`) corresponding to **inout** and **out** parameters of the original operation plus one additional parameter if the operation has a result value and an additional last parameter that is an Integer. The signal is received as:

  ```
  pREPLY(aINOUTpar,newn)
  ```

  where `newn` is an implicit integer variable attribute of the active-container, each **inout** or **out** parameter of the signal is received into the corresponding parameter of the operation call, and the value of the last item in `aINOUTpar` is received in an implicit variable attribute of the active-container if the CallOperationAction returns a value.

A <<Pseudostate>> Pseudostate with kind == choice where the common left-hand operand of the guard properties of the outgoing properties is the Boolean expression `n = newn`. In both outgoing properties the effect is empty. In the outgoing property for false, the target property is the State defined above, because the instance of `pREPLY` received does not match `n`. In the outgoing property for true, the target property is the node that originally followed the <<CallOperationAction>> CallOperationAction if the operation is not used as an expression.

9.6.4 Notation

The graphical symbol for a CallOperationAction is shown in UML-SS Figure 12.66, which should contain the textual syntax for <operation application> in clause 9.20.4 for SendSignalAction.

When a CallOperationAction is defined in textual syntax (for example when nested inside a graphical symbol for calling an operation or as part an action), the grammar that should be used is the grammar defined for <operation application> in clause 9.20.4 for SendSignalAction.

9.6.5 References

SDL:	10.5	Remote procedure
	11.13.3	Procedure call
	12.1.8	Behaviour of operations
	12.2.7	Operator application

12.3.5 Value returning procedure call

UML-SS: 11.3.10 CallOperationAction (from BasicActions)

9.7 ConditionalNode

The stereotype ConditionalNode extends the metaclass ConditionalNode with multiplicity [1..1]. This stereotype is abstract.

NOTE – This abstract stereotype is introduced to ensure that every ConditionalNode is either a <<Decision>> ConditionalNode or an <<If>> ConditionalNode. As the stereotype is abstract, each instance has to be one of the concrete subtypes.

9.7.1 Attributes

No additional attributes.

9.7.2 Constraints

No additional constraints.

9.7.3 Semantics

The concrete subtypes of the stereotype ConditionalNode give its semantics.

9.7.4 Notation

The concrete subtypes of the stereotype ConditionalNode give its notation.

9.7.5 References

UML-SS: 12.3.18 ConditionalNode (from CompleteStructuredActivities, StructuredActivities)

12.3.17 Clause (from CompleteStructuredActivities, StructuredActivities)

9.8 Continue

The stereotype Continue is a concrete subtype of the stereotype OpaqueAction.

NOTE – A <<Continue>> OpaqueAction represents a continue action within a loop that causes a jump to the next iteration of the loop or termination of the loop if already in the last iteration.

9.8.1 Attributes

No additional attributes.

9.8.2 Constraints

- A <<Continue>> OpaqueAction shall have an empty input property.
- Each <<Continue>> OpaqueAction shall be within the bodyPart of a LoopNode.

9.8.3 Semantics

A <<Continue>> OpaqueAction is mapped to a *Continue-node*. The *Connector-name* is given by the *Connector-name* produced by the mapping of the containing LoopNode.

9.8.4 Notation

When a <<Continue>> OpaqueAction is defined in textual syntax (for example when nested inside a task box), the following textual notation is used:

<continue statement> ::=
continue <semicolon>

9.8.5 References

SDL: 11.14.6 Loop statement

9.9 CreateObjectAction

The stereotype CreateObjectAction extends the metaclass CreateObjectAction with multiplicity [1..1].

NOTE – A create object action is used to create instances of agents and store a reference to the created instance in a variable.

9.9.1 Attributes

No additional attributes.

9.9.2 Constraints

The classifier property shall refer to an <<ActiveClass>> Class.

NOTE – CreateObjectAction is only allowed for an <<ActiveClass>> Class because, to be useful, the created object reference needs to assign a variable, element of a variable, or a parameter. To create a <<PassiveClass>> Class object, a <create request> is used and the result can be assigned to a variable.

9.9.3 Semantics

The <<CreateObjectAction>> CreateObjectAction is mapped to a *Create-request-node* where the classifier maps to the *Agent-identifier*, followed by an *Assignment* of the *Offspring-expression* to the *Variable-identifier* from the qualifiedName of the structuralFeature property of the related AddStructuralFeatureValueAction or AddVariableValueAction.

9.9.4 Notation

A CreateObjectAction is defined in textual syntax according to the following grammar:

<active object create request> ::=
<single attribute create request> | <multiple attribute create request>

<single attribute create request> ::=
<identifier> <is assigned sign> **new** <create body>

<multiple attribute create request> ::=
<identifier> <full stop> **append**
<left parenthesis> <create body> <right parenthesis>

<create body> ::=
<identifier>

NOTE 1 – This syntax differs from [ITU-T Z.100]. An <active object create request> corresponds to a create statement or create request area in [ITU-T Z.100].

NOTE 2 – The syntax above does not allow actual parameters for the create request, which is supported by SDL, but UML-SS specifically excludes doing anything other than creating the object.

The <identifier> of a <create body> shall identify an <<Active Class>> Class for an agent type. The classifier of the <<CreateObjectAction>> CreateObjectAction references the Classifier named <identifier> in the <create body>.

A <single attribute create request> represents a <<CreateObjectAction>> CreateObjectAction that has a result that is a reference to the created agent, followed by a use of this result as the value for a related implicit AddStructuralFeatureValueAction (if <identifier> corresponds to a structural feature) or AddVariableValueAction (if <identifier> corresponds to a variable in a compound statement).

A <multiple attribute create request> as an action represents a <<CreateObjectAction>> CreateObjectAction action for the <create body>. The <<CreateObjectAction>> CreateObjectAction is followed by a related implicit <<CallOperationAction>> CallOperationAction for an **append** operation on the structural feature or variable of a compound statement represented by the <identifier> of the <multiple attribute create request>. Consequently, append has to be valid for the type of the <identifier>: for example, if the multiplicity is [0..*], append concatenates a value to the end of the string value of the feature or variable.

9.9.5 References

SDL: 11.13.2 Create

UML-SS: 11.3.5 AddStructuralFeatureValueAction (from IntermediateActions)

11.3.6 AddVariableValueAction (from StructuredActions)

11.3.16 CreateObjectAction (from IntermediateActions)

11.3.47 StructuralFeatureAction (from IntermediateActions)

9.10 Empty

The stereotype Empty is a concrete subtype of the stereotype OpaqueAction.

NOTE – An <<Empty>> OpaqueAction represents an action that does nothing.

9.10.1 Attributes

No additional attributes.

9.10.2 Constraints

- An <<ExpressionAction>> OpaqueAction shall have an empty input property.

9.10.3 Semantics

An <<Empty>> OpaqueAction is not mapped to the SDL abstract syntax.

9.10.4 Notation

When an <<Empty>> OpaqueAction is defined in textual syntax (for example when nested inside a task box), textual notation is used. The textual notation should follow the following grammar:

```
<empty statement> ::=
                    <semicolon>
```

9.10.5 References

SDL: 11.14.8 Empty statement

9.11 Decision

The stereotype Decision is a concrete subtype of the stereotype ConditionalNode.

NOTE – A <<Decision>> ConditionalNode is used to define textual switch statements and maps to a *Decision-node* in SDL. There is no graphical notation, but a Pseudostate with kind == choice (which has no textual form) also maps to a *Decision-node*.

9.11.1 Attributes

No additional attributes.

9.11.2 Constraints

- The body property of each Clause shall have exactly one element and this shall be a SequenceNode.
- The left-hand side of the expression (as defined in clause 9.12, ExpressionAction, clause 9.16, OpaqueAction and clause 10.1, Expression) of each test of each Clause shall be the same as the left-hand side of the expression of any other test of a Clause (because these all map to the same *Decision-question*), and therefore the left-hand sides of the expressions are all of the same data type.

9.11.3 Semantics

A <<Decision>> ConditionalNode is mapped to a *Decision-node*. The Clause defines the *Decision-question* and *Decision-answer-**set***. The left-hand side of any test (they are all the same) is an <<ExpressionAction>> OpaqueAction that maps to *Decision-question*. For each Clause, the operation and right-hand side of the test define the *Range-condition* in each *Decision-answer*. The body of the Clause maps to *Transition* in the corresponding *Decision-answer*.

9.11.4 Notation

When a <<Decision>> ConditionalNode is defined in textual syntax, the textual notation should follow the following grammar:

<decision statement> ::=
    switch (<question>)
    <left curly bracket> <decision statement body> <right curly bracket>

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

<algorithm answer part> ::=
    case <range condition> <colon> <statement>

<algorithm else part> ::=
    default <colon> <statement>

<question> ::=
    <expression> | <character string> | **any**

9.12 ExpressionAction

The stereotype ExpressionAction is a concrete subtype of the stereotype OpaqueAction.

NOTE – An <<ExpressionAction>> OpaqueAction represents an action that only contains an expression. This is a utility to simplify the modelling of (for example) if and decision statements.

9.12.1 Attributes

No additional attributes.

9.12.2 Constraints

- An <<ExpressionAction>> OpaqueAction shall have exactly one element in its input property and this shall be a ValuePin.
- The value property of the input property of an <<ExpressionAction>> OpaqueAction shall contain a <<ValueSpecification>> ValueSpecification that follows the rules in clause 10, ValueSpecification.

9.12.3 Semantics

An <<ExpressionAction>> OpaqueAction is mapped to an *Expression* as defined by the <<ValueSpecification>> ValueSpecification in the contained ValuePin.

9.12.4 Notation

The notation for expressions is defined in clause 10, ValueSpecification.

9.13 For

The stereotype For is a concrete subtype of the stereotype LoopNode.

NOTE – A LoopNode stereotyped by <<For>> represents a traditional programming language for loop.

9.13.1 Attributes

Stereotype attributes:

– stepGraphPart: SequenceNode [1..1]. The SequenceNode to execute after the body of the loop, normally to carry out such actions as stepping the loop variables.

9.13.2 Constraints

- The setupPart shall have exactly one executableNode element and this shall be a SequenceNode. Each executableNode of this SequenceNode shall be either an AddVariableValueAction node (to initialize variables including loop variables), or a CallOperationAction node (to invoke an operation needed before entering the loop).
- The executableNode of a stepGraphPart shall be an AddVariableValueAction or a CallOperationAction.

9.13.3 Semantics

The loopVariable maps to the *Variable-definition-**set*** of the *Compound-node*.

The setupPart maps to the *Init-graph-node* list of the *Compound-node*, defining the initialization of the loop.

The stepGraphPart maps to *Step-graph-node* list.

Otherwise, the semantics are as defined for the stereotype LoopNode.

9.13.4 Notation

When an <<For>> LoopNode is defined in textual syntax (for example when used inside a task box), the textual notation should follow the following grammar:

```
<for statement> ::=
                    for <left parenthesis> [ <for setup> ] ; [ <loop test> ] ; [ <for step> ] <right parenthesis>
                    <loop body>

<for setup> ::=
                    <for setup item> { <comma> <for setup item> }*

<for setup item> ::=
                    <local variable definition>
               |    <assignment>
               |    <operation application>

<for step> ::=
                    <for step item> { <comma> <for step item> }*

<for step item> ::=
                    <assignment>
               |    <operation application>
```

Each <local variable definition> in a <for setup item> shall include an <is assigned sign> <expression> for the variable <name> that represents the AddVariableValueAction of the setupPart. To avoid ambiguity with <assignment>, one <name> is allowed in the <local variable definition> (<comma> <name> is not permitted). The <name> corresponds to a loopVariable.

Each <assignment> in a <for setup item> represents an AddVariableValueAction of the setupPart.

Each <operation application> in a <for setup item> represents a CallOperationAction of the setupPart.

The sequence order for these setupPart actions is the order in which they occur (left to right) in the <for setup>.

Each <assignment> in a <for step item> represents an AddVariableValueAction of the stepGraphPart.

Each <operation application> in a <for step item> represents a CallOperationAction of the stepGraphPart.

The sequence order for these setupPart actions is the order in which they occur (left to right) in the <for step>.

9.13.5 References

SDL: 11.14.1 Compound Statement

11.14.6 Loop statement

9.14 If

The stereotype If is a concrete subtype of the stereotype ConditionalNode.

NOTE – An <<If>> ConditionalNode is used to define a textual if statement and maps to a *Decision-node* in SDL. There is no graphical notation, but a Pseudostate with kind == choice (which has no textual form) also maps to a *Decision-node*.

9.14.1 Attributes

No additional attributes.

9.14.2 Constraints

- An <<If>> ConditionalNode shall have either one or two Clause elements. If it has one Clause, this shall have a test that is an <<ExpressionAction>> OpaqueAction with an OpaqueExpression of Boolean type. If it has two Clause elements, it shall have one Clause that has a test that is an <<ExpressionAction>> OpaqueAction with an OpaqueExpression of type Boolean and one else clause (following the definition of 'else clause' in 12.3.11 in the UML specification) that shall have no test.
- The body of each Clause shall have exactly one element and this shall be a SequenceNode.

9.14.3 Semantics

An <<If>> ConditionalNode is mapped to a *Decision-node*. The test (an <<ExpressionAction>> OpaqueAction) in one of the clauses defines the *Expression* of the *Decision-question*. The body of this Clause defines the *Decision-answer-**set*** of the *Decision-node*. This set will only contain one *Decision-answer*. This *Decision-answer* will have a *Range-condition* representing *True* and a *Transition* that is defined by the body of this Clause.

The body of the other Clause (that shall be an else clause according to the constraints above), if present, defines the optional *Else-answer*.

9.14.4 Notation

When an <<If>> ConditionalNode is defined in textual syntax (for example when used inside a task box), the textual notation should follow the following grammar:

<if statement> ::=
if <left parenthesis> <expression> <right parenthesis> <statement>
[**else** <statement>]

The <expression> represents the test. The non-optional <statement> represents the body of the Clause with the test. If the second <statement> is present, it represents the body of the else Clause. If it is absent, this body is an <<Empty>> OpaqueAction.

9.14.5 References

SDL: 11.13.5 Decision

11.14.4 If statement

9.15 LoopNode

The stereotype LoopNode extends the metaclass LoopNode with multiplicity [1..1]. This stereotype is abstract.

NOTE – This abstract stereotype is introduced to ensure that every LoopNode is a <<For>> LoopNode or a <<While>> LoopNode and to introduce constraints that apply in both cases. As the stereotype is abstract, each instance has to be one of the concrete subtypes.

9.15.1 Attributes

No additional attributes.

9.15.2 Constraints

General class constraints that apply to the stereotypes <<For>> and <<While>> in this profile are:

- A LoopNode shall have a name.
- The isTestedFirst attribute shall be true.
- The bodyPart shall have exactly one element and this shall be a SequenceNode.
- The test shall have one element that is an <<ExpressionAction>> OpaqueAction with an OpaqueExpression of Boolean type.
- The result property shall be empty.
- The bodyOutput property shall be empty.
- The loopVariableInput property shall be empty.

9.15.3 Semantics

A LoopNode maps to a *Compound-node*. The name of the LoopNode maps to the *Connector-name* of the *Compound-node*.

The *Transition* of the *Compound-node* is a *Decision-node*. The test property maps to the *Decision-question* and the first executableNode element of the bodyPart property maps to the *Transition* part of the *Decision-answer*. The *Range-condition* part of the *Decision-answer* is always a representation of the range condition "==true". The *Decision-node* has an *Else-answer* that consists of *Break-node* with a *Connector-name* that is the same as the *Connector-name* of the *Compound-node*.

NOTE – After the *Decision-node* has been interpreted, the *Compound-node* behaviour is to interpret the *Step-graph-node* list followed by re-interpretation of the *Transition* in a loop. The loop is terminated if the *Else-answer* is reached, or if either the *Decision-answer* or *Step-graph-node* list terminates the loop.

Its concrete subtypes give additional semantics.

9.15.4 Notation

The syntax for loops given in this profile is given by <for statement> and <while statement> in the notation for If stereotype and While stereotype respectively. The following are common elements:

```
<loop test> ::=
                        <expression>

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

The <loop test> represents the test.

The <loop body> represents the bodyPart.

NOTE – The loop statement syntax of [ITU-T Z.100] is not supported.

9.15.5 References

SDL: 11.14.1 Compound Statement

11.14.6 Loop statement

UML-SS: 12.3.35 LoopNode (from CompleteStructuredActivities, StructuredActivities)

9.16 OpaqueAction

The stereotype OpaqueAction extends the metaclass OpaqueAction with multiplicity [1..1]. This stereotype is abstract.

NOTE – This abstract stereotype is introduced to ensure that every OpaqueAction is one of the subtypes: <<Break>> OpaqueAction or <<Continue>> OpaqueAction or <<Empty>> OpaqueAction or <<ExpressionAction>> OpaqueAction or <<ResetAction>> OpaqueAction or <<SetAction>> OpaqueAction. As the stereotype is abstract, each instance has to be one of the concrete subtypes.

9.16.1 Attributes

No additional attributes.

9.16.2 Constraints

No additional constraints.

9.16.3 Semantics

The concrete subtypes of the stereotype OpaqueAction give its semantics.

9.16.4 Notation

The concrete subtypes of the stereotype OpaqueAction give its notation.

9.16.5 References

UML-SS: 11.3.26 OpaqueAction (from BasicActions)

9.17 ResetAction

A timer is cancelled with a reset action represented by a ResetAction stereotype. The ResetAction stereotype is a concrete subtype of the stereotype OpaqueAction.

NOTE – The reset action cancels a timer and removes any corresponding timer signals that are queued for the agent instance executing the timer.

9.17.1 Attributes

The stereotype has the following attributes:

– parameterlist: **part** ValueSpecification [*]. The expressions that correspond to the actual parameters of the timer.

– timer: Signal: The <<Timer>> Signal that represents the timer that is started by the action.

9.17.2 Constraints

• Each item in the parameterlist shall match the corresponding ownedAttribute of the timer.

9.17.3 Semantics

A <<ResetAction>> OpaqueAction is mapped to a *Reset-node*. The timer maps to the *Timer-Identifier*. The parameterlist maps to the *Expression* list.

9.17.4 Notation

The syntax for the reset actions is as follows:

<reset> ::=
reset <identifier> [<left parenthesis> <expression list> <right parenthesis>]

The <identifier> identifies the timer. The <expression list> is the parameterlist.

9.17.5 References

SDL: 11.15 Timer

9.18 Return

The stereotype Return is a concrete subtype of the stereotype ActivityFinalNode.

NOTE – A return represents the action to return from a procedure (in the SDL abstract grammar) to the point where the procedure was called.

9.18.1 Attributes

Stereotype attributes:

– value: OpaqueAction [0..1] An <<ExpressionAction>> OpaqueAction that represents the return value of the operation.

9.18.2 Constraints

- The <<Return>> ActivityFinalNode shall be part of an <<Activity>> Activity that is used to define the behaviour associated with an <<Operation>> Operation.
- The value shall be empty if the <<Operation>> Operation does not return a value. Otherwise, the value shall match the return type of the <<Operation>> Operation.
- The OpaqueAction in the value property shall be an <<ExpressionAction>> OpaqueAction.

9.18.3 Semantics

A <<Return>> ActivityFinalNode is mapped to an *Action-return-node* if the value property is empty, otherwise to a *Value-return-node*. If it is mapped to a *Value-return-node*, the value property defines the *Expression* in the *Value-return-node*.

9.18.4 Notation

When a <<Return>> ActivityFinalNode is defined in textual syntax (for example when used inside a task box), the textual notation should follow the following grammar:

<return statement> ::=
return [<return body>] <semicolon>

<return body> ::=
<expression>

If <<Return>> ActivityFinalNode is shown graphically, the UML notation is used with the addition of the <return body> (if there is one) close to the symbol.

The <expression> gives the value of the <<Return>> ActivityFinalNode.

9.18.5 References

SDL: 11.12.2.4 Return statement

9.19 SequenceNode

The stereotype SequenceNode extends the metaclass SequenceNode with multiplicity [1..1].

NOTE – A sequence node is a sequence of actions and is either a node of an activity or describes the body of a compound node.

9.19.1 Attributes

No additional attributes.

9.19.2 Constraints

- Each ExecutableNode that is an executableNode property of a SequenceNode and is an Action shall be an AddStructuralFeatureValueAction or an AddVariableValueAction or a CallOperationAction or a CreateObjectAction or an OpaqueAction or a SendSignalAction.
- Each ExecutableNode that is an executableNode property of a SequenceNode and is a StructuredActivityNode shall be a ConditionalNode or a LoopNode.

9.19.3 Semantics

A <<SequenceNode>> SequenceNode that is the node of an Activity is mapped as described in clause 9.1.

A <<SequenceNode>> SequenceNode that is not a node of an Activity is mapped to a *Compound-node*. The variable definitions contained in the variable property of the SequenceNode map to the *Variable-definition-**set*** of the *Compound-node*. The multiplicity of a variable is mapped to a type in the same way as in a Property (see clause 7.12, Property). The actions contained in the executableNode property of the SequenceNode map to the various *Graph-node*s in the *Transition* that are contained in the *Compound-node*. The name of the <<SequenceNode>> SequenceNode defines the *Connector-name* of the *Compound-node*.

9.19.4 Notation

NOTE – The UML-SS document does not define syntax for a SequenceNode. In this profile, SequenceNode is defined using textual syntax (for example when showing a sequence of actions in an action symbol) as follows:

```
<compound statement> ::=
                    <left curly bracket> <statement list> <right curly bracket>

<statement list> ::=
                    <variable definition>* <statement>*

<variable definition> ::=
                    <local variable definition> <semicolon>

<local variable definition> ::=
                    [ <aggregation kind> ] <identifier> [ <multiplicity>]
                         <name> [ <is assigned sign> <expression> ]
                         { <comma> <name> [ <is assigned sign> <expression> ] }*

<aggregation kind> ::=
                    part

<statement> ::=
                    <empty statement>
               |    <compound statement>
               |    <algorithm action statement>
               |    <if statement>
               |    <decision statement>
               |    <while statement>
               |    <for statement>
               |    <terminating statement>
               |    <labelled statement>

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

<algorithm action statement> ::=
<output> <semicolon>
| <active object create request> <semicolon>
| <set> <semicolon>
| <reset> <semicolon>

<labelled statement> ::=
<name> <colon> <statement>

If the statement is labelled (a <labelled statement>), the <name> is the name of the corresponding statement (the CreateObjectAction, SendSignalAction, CallOperationAction, AddVariableValueAction, AddStructuralFeatureValueAction, OpaqueAction, ConditionalNode, LoopNode or ActivityFinalNode represented by <statement> of <labelled statement>); otherwise, the name is given an anonymous unique name.

9.19.5 References

SDL: 11.14.1 Compound statement

UML-SS: 12.3.47 SequenceNode (from StructuredActivities)

9.20 SendSignalAction

The stereotype SendSignalAction extends the metaclass SendSignalAction with multiplicity [1..1].

NOTE – A send signal action outputs a signal from the executing agent, optionally specifying the target agent and the port used to send the signal.

9.20.1 Attributes

No additional attributes.

9.20.2 Constraints

The target property shall reference a ValuePin.

The onPort property shall reference a Port of the container <<ActiveClass>> Class of the <<SendSignalAction>> SendSignalAction.

9.20.3 Semantics

A <<SendSignalAction>> SendSignalAction is mapped to an *Output-node*. The qualifiedName of signal property maps to the *Signal-identifier*. The target property maps to the *Signal-destination*. The onPort property maps to the *Direct-via*. The argument property maps to the *Expression* list.

9.20.4 Notation

UML standard notation is used to show a SendSignalAction in a transition oriented statemachine syntax with the text content of the symbol being as in <output body> below. For an example, see UML-SS Figure 15.44 – Symbols for Signal Receipt, Sending and Actions on transition.

When a SendSignalAction is defined in textual syntax (for example, when nested inside an action sequence symbol), textual notation as defined in the following grammar should be used:

<output> ::=
output <output body>
| <circumflex accent> <output body>

NOTE 1 – The <circumflex accent> is an extension compared with Z.100 syntax. It is not necessary for a tool to support both alternatives.

<output body> ::=
<operation application> { <comma> <operation application> }*

The following syntax is used for both SendSignalAction and in clause 9.6.4 CallOperationAction and therefore the <type expression> represents the signal property (of <<SendSignalAction>>

SendSignalAction) or the operation property (of <<CallOperationAction>> CallOperationAction) respectively.

<operation application> ::=
<operator application>
| <method application>

<operator application> ::=
<type expression> [<actual parameters>] <communication constraints>

<method application> ::=
<primary> <full stop> <type expression> [<actual parameters>]
<communication constraints>

<type expression> ::=
<type identifier>

NOTE 2 – The use of <primary> in a <method application> of an <output body> is explained with <communication constraints> below.

<actual parameters> ::=
<left parenthesis> <actual parameter list> <right parenthesis>

<actual parameter list> ::=
[<expression>] { <comma> [<expression>] }*

The optional <actual parameters> represent the argument list. If the <actual parameters> are omitted, the list is empty.

<communication constraints> ::=
[<via path>]

<via path> ::=
via <identifier>

A <via path> represents the onPort and, if omitted, the onPort is empty.

NOTE 3 – To specify a specific destination, the <method application> syntax is used, where the <primary> specifies the destination. By comparison, SDL allows a destination to be given in <communication constraints>.

The <primary> of a <method application> used as an <output body> represents the target and shall not be a <literal>. The <primary> shall be an <operation application>, a bracketed <expression>, an <extended primary>, or an <active primary> that is a reference to an element of an instance of an <<ActiveClass>> Class (an agent instance) or a reference to an instance of an <<ActiveClass>> Class (an agent instance set). A **self**, **parent**, **offspring** or **sender** <pid expression> is a valid <primary>.

9.20.5 References

SDL: 11.13.4 Output

UML-SS: 11.3.45 SendSignalAction (from BasicActions)

NOTE – The syntax for <range condition> is given in clause 7.12.4, the notation for Property.

The <question> represents the left-hand side of every test expression (the question). The <range condition> of each <algorithm answer part> determines the operator for the expression and the value for the right-hand side of the expression. If the <range condition> consists of a single <open range>, the operator is the operator corresponding to the <equality sign>, <not equals sign>, <less than sign>, <greater than sign>, <less than or equals sign>, or <greater than or equals sign>. Otherwise, <range condition> is evaluated to a set value that contains the values specified by the <range condition> and the operator is the membership operator for the left-hand side of the test being in this set. The type of the set is the set of all possible values of the type of the left-hand side of the test. The <statement> of the <algorithm answer part> represents the body of the Clause with the test. The test for the <algorithm else part> is the question not being in the set of values covered by any of the right-hand sides (that is the test is true only if all other tests are false). The <statement> of the <algorithm else part> represents the body of the else Clause.

9.20.6 References

SDL: 11.13.5 Decision

11.14.5 Decision statement

9.21 SetAction

A timer is started with a set action represented by a SetAction stereotype. The SetAction stereotype is a concrete subtype of the stereotype OpaqueAction.

NOTE – The set action gives a timer an expiry time.

9.21.1 Attributes

The stereotype has the following attributes:

–	parameterlist: **part** ValueSpecification [*]	The expressions that correspond to the actual parameters of the timer.
–	timer: Signal	The <<Timer>> Signal that represents the timer that is started by the action.
–	timeExpression: ValueSpecification	The duration that determines when the timer will expire.

9.21.2 Constraints

- Each item in the parameterlist shall match the corresponding ownedAttribute of the timer.
- The timeExpression shall be of the Time type.

9.21.3 Semantics

A <<SetAction>> OpaqueAction is mapped to a *Set-node*. The timer maps to the *Timer-Identifier*. The parameterlist maps to the *Expression* list and timeExpression maps to *Time-expression*.

9.21.4 Notation

The syntax for the set actions is as follows:

<set> ::=

set <identifier> [**(** <expression list> **)**] [<is assigned sign> <expression>]

The <identifier> identifies the timer. The <expression list> is the parameterlist. If <is assigned sign> <expression> is omitted, the timeExpression is set to **now** + the default duration of the timer. Otherwise, the <expression> gives the timeExpression.

9.21.5 References

SDL: 11.15 Timer

9.22 Stop

The stereotype Stop is a concrete subtype of the stereotype ActivityFinalNode.

NOTE – A stop represents the action to terminate the enclosing <<ActiveClass>> Class instance (the enclosing agent).

9.22.1 Attributes

No additional attributes.

9.22.2 Constraints

No additional constraints.

9.22.3 Semantics

A <<Stop>> ActivityFinalNode is mapped to a *Stop-node*.

9.22.4 Notation

The <<Stop>> ActivityFinalNode may only be used in textual syntax for transitions. The textual notation should adhere to the following grammar:

```
<stop statement> ::=
                stop <semicolon>
```

9.22.5 References

SDL: 11.12.2.3 Stop statement

9.23 While

The stereotype While is a concrete subtype of the stereotype LoopNode.

NOTE – A LoopNode stereotyped by <<While>> represents a traditional programming language while loop.

9.23.1 Attributes

No additional attributes.

9.23.2 Constraints

- The setupPart property of a <<While>> LoopNode shall be empty.
- The loopVariable property of a <<While>> LoopNode shall be empty.

9.23.3 Semantics

The *Variable-definition-**set*** is empty.

The *Init-graph-node* list is empty.

The *Step-graph-node* list is empty.

Otherwise, the semantics are as defined for the stereotype LoopNode.

9.23.4 Notation

When a <<While>> LoopNode is defined in textual syntax (for example when used inside a task box), the textual notation should follow the following grammar:

```
<while statement> ::=
                while <left parenthesis> <loop test> <right parenthesis> <loop body>
```

The relationship to the meta-model elements is defined in clause 9.15, LoopNode.

9.23.5 References

SDL: 11.14.1 Compound Statement

11.14.6 Loop statement

10 ValueSpecification

A value is specified as a non-terminal expression or a literal for one of the values of a primitive type or a reference to an object that contains a value. An expression is a node in an expression tree that has a number (possibly zero) of operands that themselves specify values and therefore can be expressions, literals or instance values. A value is represented textually and the syntax is a concrete textual syntax based on SDL. Consequently, the components of an expression in SDL-UML usually have a one-to-one correspondence with respective SDL abstract syntax items that would result from analysing the text as SDL.

The following metaclasses from the UML Kernel package are included:

– Expression
– InstanceValue
– LiteralInteger
– LiteralNull
– LiteralString
– LiteralUnlimitedNatural
– ValueSpecification

10.1 Expression

The stereotype Expression extends the metaclass Expression with multiplicity [1..1].

NOTE – An expression is a value specification that has the logical form of an operator with operands, though the concrete syntax may be some other form, such as a conditional expression. The leaf node operand of an expression is an expression operator that has no operands or a literal specification or an instance value.

10.1.1 Attributes

Stereotype attributes:

– isConstant: Boolean true if the expression is a constant expression. This is a derived attribute.

The stereotypes that extend expressions and their attributes are defined in the context of the concrete syntax given in the Notation clause below.

10.1.2 Constraints

• The <<PassiveClass>> Class for a create request shall have an operator with the name Make that has, as a result, an instance of the <<PassiveClass>> Class.
• In the operand list of an <<Expression>> Expression that is mapped to an *Operation-application*, each operand shall match the type of the corresponding parameter of the operation.
• In the operand list of an <<Expression>> Expression that is mapped to a *Conditional-expression*, the first operand shall be a Boolean and each of the other operands shall be of the same type, that is the type (of the <<Expression>> Expression).
• The type (of the <<Expression>> Expression) shall match the type required in the context of the <<Expression>> Expression.

10.1.3 Semantics

The isConstant property is false if the <<Expression>> Expression has an operand that is an <<Expression>> Expression with isConstant false or is an <<InstanceValue>> InstanceValue that maps to a *Variable-access*. In all other cases, isConstant is true.

An <<Expression>> Expression is mapped to an *Expression*. The operand order is defined by the order in which the operands appear in the concrete syntax (left to right) except where explicitly noted below.

The symbol of an <<Expression>> Expression is a String. Where the symbol represents an infix operator, the text string is <quotation mark> infix-operation-name <quotation mark> qualified by the definition context of the operator. For example, the symbol for <implies sign> is the text string

"=>" (including the quotation marks) normally qualified by the package for predefined data. Where the symbol represents a monadic prefix operator, the text string is <quotation mark> monadic-operation-name <quotation mark>. For some expressions (such as <range check expression>), the symbol is the text string for an implicit identifier derived from the textual syntax as defined below. In all other cases, the symbol is the text string given for the operation identifier of the expression with its qualifier.

An <<Expression>> Expression that is a constant expression (isConstant is true) is mapped to an *Expression* that is a *Constant-expression*. An <<Expression>> Expression that is not a constant expression (isConstant is false) is mapped to an *Expression* that is an *Active-expression*. In the following, *Expression* is used to mean *Constant-expression* or *Active-expression* depending on the value of isConstant.

Unless explicitly stated otherwise, the <<Expression>> Expression is mapped to the *Operation-application* alternative of *Expression*. When the <<Expression>> Expression is mapped to an *Expression* that is an *Operation-application*, the symbol is used to determine the *Operator-identifier* of the *Operation-application*. The operand list maps to the *Expression* list of the *Operation-application*.

An <<Expression>> Expression with the implicit unique symbol for the range check is mapped to an *Operation-application* for the range check.

An <<Expression>> Expression with the implicit unique symbol for conditional expressions is mapped to a *Conditional-expression*. The operand list maps to the *Boolean-expression*, *Consequence-expression*, and *Alternative-expression* of the *Conditional-expression*.

An <<Expression>> Expression with the symbol for **now**, **self**, **parent, offspring** or **state** is mapped to a *Now-expression*, *Self-expression*, *Parent-expression*, *Offspring-expression*, or *State-expression* respectively.

An <<Expression>> Expression with the symbol for **active** or **rem** is mapped to a *Timer-active-expression* or *Timer-remaining-expression* respectively. The first operand maps to *Timer-identifier* and the remaining operand list maps to the *Expression-list*.

An <<Expression>> Expression with the symbol for **any** is mapped to an *Any-expression* with type mapped to the *Sort-reference-identifier* of the *Any-expression*.

10.1.4 Notation

The grammar is (except where explicitly stated) a subset of the grammar from SDL.

The symbol and operand set of an <<Expression>> Expression are defined as follows:

<expression> ::=
<expression0>
| <range check expression>

<range check expression> ::=
<operand2> **in type** { <sort identifier> <constraint> | <sort identifier> }

<constraint> ::=
constants <left parenthesis> <range condition> <right parenthesis>
| <size constraint>

<size constraint> ::=
size <left parenthesis> <range condition> <right parenthesis>

The symbol in a <range check expression> is defined by an implicit identifier for the range check derived from the <constraint> or <sort identifier> of the <range check expression> as defined in clause 12.1.9.5 of [ITU-T Z.100] and the operand is the <<ValueSpecification>> ValueSpecification for the <operand2> of the <range check expression>. The range check is an Operation (with an arbitrary unique name) derived from the <constraint> of the

<range check expression> or the <constraint> of the sort identified in the <range check expression> as defined in clause 12.1.9.5 of [ITU-T Z.100] (and therefore shall be a valid <constraint> according to [ITU-T Z.100]).

<expression0> ::=
<operand>
| <create expression>
| <value returning procedure call>

<create expression> ::=
<multiple attribute create request>
<create request>

<create request> ::=
new <identifier>

NOTE 1 – The syntax for <create expression> is changed compared with SDL to use the keyword **new** instead of **create** or (in the case of a data type) Make.

The form <multiple attribute create request> shall be used when <<Active Class>> Class instances are created; otherwise, the form <create request> shall be used.

A <multiple attribute create request> is a shorthand notation for inserting a <multiple attribute create request> action just before the action where the <create expression> occurs. The variable assigned in the action replaces the create request in the original expression. If <create expression> occurs several times in an expression, one distinct variable is used for each occurrence. In this case, the order of the inserted create requests and variable assignments is the same as the order of the <create expression>s. From the transform for <create expression>, the <create expression> in the expression is replaced by a <variable access>, and therefore the <create expression> is an InstanceValue rather than an Expression.

A <create request> for a <<Passive Class>> Class or a DataType is an invocation of the Make operation for the type identified by <identifier> and symbol represents this Make. The operand list is empty.

The symbol in a <value returning procedure call> is the text string for the identity of the called procedure. The operand set is the actual parameter set of the procedure call.

<operand> ::=
<operand0>
| <operand> <implies sign> <operand0>

The symbol in an <operand> with an <implies sign> is the text string for the <implies sign> qualified by the type for the context and the operand set is the <<ValueSpecification>> ValueSpecification pair for <operand> and <operand0>.

<operand0> ::=
<operand1>
| <operand0> { **or** | **xor** } <operand1>

The symbol in an <operand0> with an '**or**' or '**xor**' is the text string for the respective operator qualified by the type for the context and the operand set is the <<ValueSpecification>> ValueSpecification pair for <operand0> and <operand1>.

<operand1> ::=
<operand2>
| <operand1> **and** <operand2>

The symbol in an <operand1> with an '**and**' is the text string for '**and**' qualified by the type for the context and the operand set is the <<ValueSpecification>> ValueSpecification pair for <operand1> and <operand2>.

```
<operand2> ::=
        <operand3>
    |   <operand2> { <greater than sign>
                   | <greater than or equals sign>
                   | <less than sign>
                   | <less than or equals sign>
                   | in } <operand3>
    |   <equality expression>
```

The symbol in an <operand2> with a <greater than sign> or <greater than or equals sign> or <less than sign> or <less than or equals sign> or '**in**' is the symbol for the respective operator qualified by the type for the context and the operand set is the <<ValueSpecification>> ValueSpecification pair for <operand2> and <operand3>.

```
<equality expression> ::=
        <operand2> { <equality sign> | <not equals sign> } <operand3>
```

The symbol in an <equality expression> with an <equality sign> or <not equals sign> is the text string for the respective sign qualified by the type for the context and the operand set is the <<ValueSpecification>> ValueSpecification pair for <operand2> and <operand3>.

```
<operand3> ::=
        <operand4>
    |   <operand3> { <plus sign> | <hyphen> | <concatenation sign> } <operand4>
```

The symbol in an <operand3> with a <plus sign> or <hyphen> or <concatenation sign> is the symbol for the respective sign qualified by the type for the context and the operand set is the <<ValueSpecification>> ValueSpecification pair for <operand3> and <operand4>.

```
<operand4> ::=
        <operand5>
    |   <operand4> { <asterisk> | <solidus> | mod | rem } <operand5>
```

The symbol in an <operand4> with an <asterisk> or <solidus> or **mod** or **rem** is the symbol for the respective sign qualified by the type for the context and the operand set is the <<ValueSpecification>> ValueSpecification pair for <operand4> and <operand5>.

```
<operand5> ::=
        [ <hyphen> | not ] <primary>
```

The symbol in an <operand5> with a <hyphen> or **not** is the symbol for the respective sign qualified by the type for the context and the operand set is the <<ValueSpecification>> ValueSpecification for the <primary>.

```
<primary> ::=
        <operation application>
    |   <literal>
    |   <left parenthesis> <expression> <right parenthesis>
    |   <conditional expression>
    |   <extended primary>
    |   <active primary>
```

NOTE 2 – The SDL synonym is not included. Instead, a read-only element should be used.

The symbol in an <operation application> is the text string for the name of the called operation qualified by the type for the context.

A bracketed <expression> is used to syntactically separate the <expression>. The representation as meta-model elements is otherwise the same as an <expression> without brackets.

```
<active primary> ::=
        <variable access>
    |   <imperative expression>
```

A <variable access> is an <<InstanceValue>> InstanceValue (see clause 10.2, InstanceValue).

<expression list> ::=

<expression> { **,** <expression> }*

An <expression list> is an <expression> list and the representation of each <expression> is treated in turn when the <expression list> is used.

<constant expression> ::=

<constant expression0>

A <constant expression> is an expression that does not contain an <active primary> or a <value returning procedure call>. It is treated as an <expression>.

From SDL: 12.2.6 Conditional expression

<conditional expression> ::=

<Boolean expression>
<question mark> <consequence expression>
<colon> <alternative expression>

NOTE 3 – Conditional expressions use a different syntax from [ITU-T Z.100].

<consequence expression> ::=

<expression>

<alternative expression> ::=

<expression>

The <Boolean expression> shall not be a <conditional expression>.

The symbol for a <conditional expression> represents the name of the implicit operation for the conditional expression. The operand list consists of three <<ValueSpecification>> ValueSpecification items for the <Boolean expression>, <consequence expression>, and <alternative expression>.

From SDL: 12.2.4 Extended primary

<extended primary> ::=

<indexed primary>
| <field primary>
| <composite primary>

<indexed primary> ::=

<primary> <left square bracket> <actual parameter list> <right square bracket>

Although the syntax of <actual parameter list> allows each parameter <expression> to be omitted, it is not allowed to omit any parameter <expression> (that is, actual parameters are not allowed to be undefined).

The symbol for an <indexed primary> represents the name of the Extract operation for the type of the <primary>. The operand list is the <<ValueSpecification>> ValueSpecification for the <primary> followed by <<ValueSpecification>> ValueSpecification list for the <actual parameter list>. The number of index values and the type of each value shall be consistent with the definition of the Extract operation for the type of the <primary>.

<field primary> ::=

<primary> <full stop> <field name> <communication constraints>
| <field name>

<field name> ::=

<name>

If <primary> identifies an <<ActiveClass>> Class instance, <field primary> corresponds to an exchange of implicit signals to import the value of the primary from the active class. The <field name> shall be the name of a variable part of the type of the <primary>. A non-empty <communication constraints> is valid in <field primary> only if the field is part of an <<ActiveClass>> Class.

The alternative of <field primary> starting <field name> is only valid in a method body and is a shorthand form for **this** <full stop> <field name> <communication constraints>. The <field name> shall be the name of a field for the type of the method.

If the field is not part of an <<ActiveClass>> Class, the symbol identifies the field Extract operator for the field and the operand is the <<ValueSpecification>> ValueSpecification for the <primary> or the implicit parameter (**this**) for the target of the method.

If the field is part of another <<ActiveClass>> Class instance, there is an implicit value returning procedure to return the value. The <field primary> is transformed to <type expression> <communication constraints> for an <operator application> that is treated as a <value returning procedure call> (see below), where <type expression> identifies the implicit operator that does the remote access.

<composite primary> ::=
<identifier> <composite begin sign> <actual parameter list> <composite end sign>

The <composite primary> has a mandatory <identifier> (instead of the optional qualifier in SDL) to identify the type of the created value. The symbol identifies the Make operator of the type identified by the <identifier> and <actual parameter list> represents the operand list. A <composite primary> is only valid if Make is defined for the type identified by the <identifier>, which acts as a qualifier for the name of Make.

The syntax from SDL 11.13.2 for <actual parameter list> is given in clause 9.20, SendSignalAction.

<imperative expression> ::=
<now expression>
| <pid expression>
| <timer active expression>
| <timer remaining duration>
| <any expression>
| <state expression>

<now expression> ::=
now

<pid expression> ::=
self
| **parent**
| **offspring**
| **sender**

<timer active expression> ::=
active <left parenthesis> <timer identifier>
[<left parenthesis> <expression list> <right parenthesis>]
<right parenthesis>

<timer remaining duration> ::=
rem <left parenthesis> <timer identifier>
[<left parenthesis> <expression list> <right parenthesis>]
<right parenthesis>

<any expression> ::=
any <left parenthesis> <identifier> <right parenthesis>

The <identifier> of <any expression> shall identify a passive class or a data type.

<state expression> ::=
state

For each <imperative expression>, the symbol represents an unique identifier for an implicit operator for that expression. For **now**, **self**, **parent, offspring** and **state**, the operand list is empty. For **active** and **rem**, the operand list is the identified timer followed by the expression list where each of the expressions shall be compatible with the timer parameters.

For **any**, the operand list is empty but the type of <<Expression>> Expression is set to the type identified by <identifier>.

<value returning procedure call> ::=
[**call**] [**this**] <operator application>

This syntax is modified compared with SDL to use <operator application>.

The optional keyword **call** is to allow a procedure call to be distinguished from other syntactically equivalent items with the same signature.

If **this** is used, operator identifier (in the <type expression> of <operator application>) shall denote an enclosing operation, and if the operation is specialized the identifier of the specialized operation replaces this identifier. When **this** is not used, no substitution takes place so the enclosing operation is called. In all other respects, a <value returning procedure call> is treated as a <<CallOperationAction>> CallOperationAction.

10.1.5 References

SDL: 12.2.1 Expressions

UML-SS: 7.3.18 Expression (from Kernel)

10.2 InstanceValue

The stereotype InstanceValue extends the metaclass InstanceValue with multiplicity [1..1].

NOTE – An instance value is a value specification that is the result of accessing a structural feature that maps to a *Variable-definition* or *Parameter* in SDL, or is a literal for an enumerated type.

10.2.1 Attributes

No additional attributes.

10.2.2 Constraints

No additional constraints.

10.2.3 Semantics

If the instance is an InstanceSpecification for an Enumeration (that is the <literal> denotes an enumeration value such as a value for `Character`, `Real`, `Bit`, `Bitstring`, or a defined enumeration), the qualifiedName of the <<InstanceValue>> InstanceValue is mapped to a *Literal*. Otherwise, the qualifiedName of the <<InstanceValue>> InstanceValue is mapped to a *Variable-access*.

10.2.4 Notation

From SDL: 12.3.2 2 Variable access

<variable access> ::=
<identifier>
| **this**

If **this** is used, the <variable access> shall be in the body of a method and **this** denotes the object the method operates on.

The syntax for <literal> differs from the syntax in SDL 12.2.2, because in SDL-UML the lexical units for string, real, and integer literal names and characters are distinct from other names, whereas in SDL these are all treated as names.

<literal> ::=
<literal identifier>
| <string name>
| <real name>
| <integer name>
| <bit string>
| <hex string>
| <character>

<string name> ::=
<character string>

NOTE 1 – The syntax for <identifier> is given in clause 5.2, Names and name resolution: NamedElement, and the lexical tokens <character string>, <real name>, <integer name>, <bit string>, <hex string>, and <character> are described in clause 11, Lexical Rules.

A <string name> represents a LiteralString not an <<InstanceValue>> InstanceValue.

If the <real name> is in a context that requires a `Duration` value, the <real name> is transformed into a call of the implicit protected operation `duration` of `Duration` with the <real name> as a parameter and then handled as an <expression>. If the <real name> is in a context that requires a `Time` value, the <real name> is transformed into a call of the implicit protected operation `time` of `Time` with the <real name> as a parameter and then handled as an <expression>. Because `time` requires a `Duration` value, an implicit call of `duration` is invoked. Otherwise a <real name> represents a literal value of the predefined `Real` enumeration data type and specializations of that data type.

An <integer name> represents a LiteralInteger or LiteralNatural (depending on context) not an <<InstanceValue>> InstanceValue.

If the context where a <bit string> or <hex string> occurs requires an `Integer` or `UnlimitedNatural` value, the <bit string> or <hex string> is transformed to an <expression> that is a call of the `num` operation of `Bitstring` with the <bit string> or <hex string> literal as a parameter. If the context where a <bit string> or <hex string> occurs requires an `OctetString` value, the <bit string> or <hex string> is transformed to an <expression> that is a call of the `octetstring` operation of `Octetstring` with the <bit string> or <hex string> literal as a parameter. If the context where a <bit string> or <hex string> occurs requires an `Octet` value, the <bit string> or <hex string> is transformed to an <expression> that is a call of the `Octet` operation of `Octet` with the <bit string> or <hex string> literal as a parameter. If the context where a <bit string> or <hex string> occurs requires a `Bit` value and the <bit string> is '0'B or '1'B, the <bit string> represents a literal value of the predefined `Bit` data type and specializations of that data type. Otherwise, a <bit string> or <hex string> represents a literal value of the predefined `Bitstring` data type and specializations of that data type.

A <character> represents a literal value of the predefined enumeration Character data type and specializations of that data type.

If <identifier> has a name part that corresponds to one of the names defined by the predefined enumeration Character data type (NUL, SOH … IS1) and is unqualified or is qualified by Character (or a specialization of it), the <identifier> represents literal value of this data type.

NOTE 2 – It is suggested never to use the names defined by the predefined enumeration Character data type (NUL, SOH … IS1) except to represent Character values. If used for the name of some other item, such a name should always be qualified.

If the <identifier> has a name part that is 'true' or 'false' and is unqualified or is qualified by the predefined Boolean data type (or a specialization of it), the <identifier> represents a <<LiteralBoolean>> LiteralBoolean not an InstanceValue.

If the <identifier> has a name part that is 'NULL', the <identifier> represents a LiteralNull not an <<InstanceValue>> InstanceValue.

10.2.5 References

SDL:	12.2.2	Literal
	12.3.2	Variable access
UML-SS:	7.3.23	InstanceValue (from Kernel)

10.3 LiteralBoolean

The stereotype LiteralBoolean extends the metaclass LiteralBoolean with multiplicity [1..1].

NOTE – LiteralBoolean is used to represent the Boolean values 'true' and 'false'.

10.3.1 Attributes

No additional attributes.

10.3.2 Constraints

No additional constraints.

10.3.3 Semantics

The value of the <<LiteralBoolean>> LiteralBoolean is mapped to a *Literal* denoting the literals *true* and *false* in the predefined Boolean data type.

10.3.4 Notation

The notation is defined in clause 10.2.4.

10.3.5 References

SDL:	12.2.2	Literal
UML-SS:	7.3.26	LiteralBoolean (from Kernel)

10.4 LiteralInteger

The stereotype LiteralInteger extends the metaclass LiteralInteger with multiplicity [1..1].

NOTE – A literal integer is denoted by an <integer name> and represents an integer value.

10.4.1 Attributes

No additional attributes.

10.4.2 Constraints

No additional constraints.

10.4.3 Semantics

The value of the <<LiteralInteger>> LiteralInteger is mapped to the *Literal* for the digit sequence for the value of the integer.

10.4.5 Notation

An <integer name> represents a <<LiteralInteger>> LiteralInteger except if the context requires an `UnlimitedNatural`. The <integer name> determines the integer value.

10.4.6 References

SDL: 12.2.2 Literal

UML-SS: 7.3.27 LiteralInteger (from Kernel)

10.5 LiteralString

The stereotype LiteralString extends the metaclass LiteralString with multiplicity [1..1].

NOTE – A literal string is denoted by a <string name> and represents a string value.

10.5.1 Attributes

No additional attributes.

10.5.2 Constraints

No additional constraints.

10.5.3 Semantics

The value of the <<LiteralString>> LiteralString is mapped to the *Literal* for the string.

10.5.4 Notation

A <string name> represents a <<LiteralString>> LiteralString and determines its value.

10.5.5 References

SDL: 12.2.2 Literal

UML-SS: 7.3.30 LiteralString (from Kernel)

10.6 LiteralUnlimitedNatural

The stereotype LiteralUnlimitedNatural extends the metaclass LiteralUnlimitedNatural with multiplicity [1..1].

NOTE – A natural number is denoted by the <integer name> and this represents the value.

10.6.1 Attributes

No additional attributes.

10.6.2 Constraints

No additional constraints.

10.6.3 Semantics

The value of the <<LiteralUnlimitedNatural>> LiteralUnlimitedNatural is mapped to the *Literal* for the digit sequence for the value of the integer.

10.6.4 Notation

If the context requires an `UnlimitedNatural`, an <integer name> represents a <<LiteralUnlimitedNatural>> LiteralUnlimitedNatural. The <integer name> determines the natural value.

10.6.5 References

SDL: 12.2.2 Literal

UML-SS: 7.3.31 LiteralUnlimitedNatural (from Kernel)

10.7 LiteralNull

The stereotype LiteralNull extends the metaclass LiteralNull with multiplicity [1..1].

NOTE – The literal null denotes a value of a reference type used when no object is referenced.

10.7.1 Attributes

No additional attributes.

10.7.2 Constraints

No additional constraints.

10.7.3 Semantics

The <<LiteralNull>> LiteralNull is mapped to the *Operation-application* of the parameterless operator Null appropriate for the context where it is used.

10.7.4 Notation

The notation is defined in clause 10.2.4.

10.7.5 References

SDL: 12.1.5 Any

UML-SS: 7.3.28 LiteralNull (from Kernel)

10.8 ValueSpecification

The stereotype ValueSpecification extends the metaclass ValueSpecification with multiplicity [1..1].

NOTE – A value specification gives the description of a value that is evaluated by some action.

10.8.1 Attributes

No additional attributes.

10.8.2 Constraints

A ValueSpecification shall not be an OpaqueExpression.

NOTE – A ValueSpecification that is not a LiteralSpecification or InstanceValue could either have been treated by default as an OpaqueExpression (with [ITU-T Z.100] as the expression language attribute) or as an Expression. Expression was chosen so that the operators and operands are visible and resolved at the SDL-UML level rather than be hidden within a (Z.100 language) OpaqueExpression.

10.8.3 Semantics

A <<ValueSpecification>> ValueSpecification is an Expression or InstanceValue or LiteralSpecification, and the mapping to the abstract grammar is determined by these metaclasses.

10.8.4 Notation

The notation for a ValueSpecification is an <expression> as defined in clause 10.1, Expression.

References

SDL: 12.2.1 Expression

UML-SS: 7.3.54 ValueSpecification (from Kernel)

11 Lexical rules

The following production rules define the lexical structure of SDL-UML text definitions.

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

NOTE 1 – The syntax rules for <name>, <alphanumeric>, <letter>, <uppercase letter>, <lowercase letter> and <decimal digit> are given in clause 5.2.

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

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

<quoted name> ::=
            <apostrophe> { <quoted name character><quoted name character>+
            | <reverse solidus><any character except btnfr> } <apostrophe>
            | <apostrophe> <apostrophe>
```

The second form is used to represent quoted name consisting from only one character and to avoid its misinterpretation as a <character>.

The third form is used to represent the omission of a name, which typically is not allowed from a semantic point of view but is accepted in the syntax.

```
<quoted name character> ::=
                    <any character except apostrophe and reverse solidus>
                |   <reverse solidus> <any character>

<character string> ::=
                <quotation mark> <charstring character>* <quotation mark>

<charstring character> ::=
                    <any character except quotation mark and reverse solidus>
                |       <reverse solidus>
                    {   <reverse solidus>
                        | <quotation mark>
                        | <quotation mark> b <quotation mark>
                        | <quotation mark> t <quotation mark>
                        | <quotation mark> n <quotation mark>
                        | <quotation mark> f <quotation mark>
                        | <quotation mark> r <quotation mark>
                    }

<any character> ::=
            Any Unicode character except bell, form feed, newline, carriage return and tab.

<any character except quotation mark and reverse solidus> ::=
            Any Unicode character except
                quotation mark, reverse solidus, bell, form feed, newline, carriage return and tab.

<any character except apostrophe and reverse solidus> ::=
            Any Unicode character except
                apostrophe, reverse solidus, bell, form feed, newline, carriage return and tab.
```

<any character except btnfr> ::=
Any Unicode character except the characters 'b', 't', 'n', 'f', 'r', apostrophe and reverse solidus.

<character> ::=
<apostrophe>
{ <any character except apostrophe and reverse solidus>
| <reverse solidus>
{ <apostrophe>
| <reverse solidus>
| <quotation mark> **b** <quotation mark>
| <quotation mark> **t** <quotation mark>
| <quotation mark> **n** <quotation mark>
| <quotation mark> **f** <quotation mark>
| <quotation mark> **r** <quotation mark>
}
}
<apostrophe>

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

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

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

<note text> ::=
{ <not asterisk or solidus>
| <asterisk>+ <not asterisk or solidus>
| <solidus> }*

<not asterisk or solidus> ::=
Any Unicode character except asterisk and solidus

<composite special> ::=
<result sign>
<range sign>
| <composite begin sign>
| <composite end sign>
| <concatenation sign>
| <history dash sign>
| <greater than or equals sign>
| <implies sign>
| <less than or equals sign>
| <not equals sign>
| <name separator>
| <equality sign>

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

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

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

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

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

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

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

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

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

NOTE 2 – The lexical unit <is assigned sign> differs from SDL where <colon> <equals sign> is used for assignment and a single <equals sign> is used for the <equality sign>, and <is assigned sign> is an alternative for <composite special>.

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

<equality sign> ::=
<equals sign> <equals sign>

NOTE 3 – The lexical unit <equality sign> is added (as compared with SDL) where a single <equals sign> is used for equality.

<not equals sign> ::=
<exclamation mark> <equals sign>

NOTE 4 – The lexical unit <not equals sign> differs from SDL where it is defined as <solidus> <equals sign>.

<name separator> ::=
<colon> <colon>

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

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

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

<exclamation mark> ::= **!**

<quotation mark> ::= **"**

<left parenthesis> ::= **(**

<right parenthesis> ::= **)**

<asterisk> ::= *

<plus sign> ::= **+**

<comma> ::= **,**

<hyphen> ::= **-**

<full stop> ::= **.**

<solidus> ::= **/**

<colon> ::= **:**

<semicolon> ::= **;**

<less than sign> ::= **<**

<equals sign> ::= **=**

<greater than sign> ::= **>**

<left square bracket> ::= **[**

<right square bracket> ::= **]**

<left curly bracket> ::= **{**

<right curly bracket> ::= **}**

<number sign> ::= **#**

<dollar sign> ::= **$**

<percent sign> ::= **%**

<ampersand> ::= **&**

<apostrophe> ::= **'**

<question mark> ::= **?**

<commercial at> ::= **@**

<reverse solidus> ::= ****

<circumflex accent> ::= **^**

<underline> ::= **_**

<grave accent> ::= **`**

<vertical line> ::= **|**

<tilde> ::= **~**

<keyword> ::=
and | **any** | **case** | **else** | **for** | **if** | **in** | **mod** | **neg** | **new** | **now** | **offspring** | **or** | **parent** | **return** | **self** | **sender** | **set** | **stop** | **switch | this** | **via** | **while** | **xor**

<space> ::=

The lexical unit <space> represents any non-printing white space character.

12 Predefined data

This clause defines a set of predefined data types as a SDL-UML library. The data types in this package are implicitly available in models constructed using this profile.

The predefined data types are divided into unparameterized types, which can be used directly and template types, which are parameterized and need to have all their parameters bound before they can be used.

The semantics of the data types and operations defined in this clause are given by mapping to the type with the same name or operator with same signature in the `Predefined` package (in Annex D of [ITU-T Z.100]), except if a different mapping is explicitly mentioned below.

In the following, signatures with an infix operation (which includes simple prefix operations) are shown as:

- "op" (type-of-left-hand-side, type-of-right-hand-side) : type-of-result

For example:

- `"`**`not`**`"(Boolean): Boolean`
- `"+" (Integer, Integer): Integer`

define infix operations for the expressions:

- **`not`** `b`

- `i + j`

where `b` is a `Boolean`, `i` and `j` are `Integer`.

Every data type has the operations "=" and "!=" defined with the signatures:

- `"=" (DataType, DataType) : Boolean`
- `"!=" (DataType, DataType) : Boolean`

therefore these are not shown below.

12.1 Unparameterized types

These are the following types: `Boolean`, `Integer`, `UnlimitedNatural`, `Character`, `String`, `Real`, `Duration`, `Time`, `Bit`, `Bitstring`, `Octet`, `Octetstring` and `Pid`.

12.1.1 Bit

`Bit` is a predefined SDL-UML DataType.

The values of `Bit` are represented by the lexical rule <bit string> literals '0'B and '1'B.

`Bit` has the following operations:

- `bit (Integer): Bit`
- `num (Bit): Integer`

12.1.2 Bitstring

`Bitstring` is a predefined SDL-UML DataType.

The values of `Bitstring` are represented by the lexical rules <bit string> and <hex string>.

`Bitstring` has the following methods:

- `length(): Integer`
- `first(): Bit`
- `last(): Bit`

`Bitstring` has the following infix operations:

- `"`**`not`**`"(Bitstring): Bitstring`
- `"`**`and`**`"(Bitstring, Bitstring): Bitstring`
- `"`**`or`**`"(Bitstring, Bitstring): Bitstring`
- `"`**`xor`**`"(Bitstring, Bitstring): Bitstring`
- `"=>"(Bitstring, Bitstring): Bitstring`
- `"+"(Bitstring, Bitstring): Bitstring` Concatenation. SDL uses //.

`Bitstring` has the following operations:

- `substring(Bitstring, Integer, Integer): Bitstring`
- `remove(Bitstring, Integer, Integer): Bitstring`
- `mkstring(Bit): Bitstring`
- `num(Bitstring): Integer`
- `bitstring(Integer): Bitstring`
- `octet(Integer): Bitstring`
- `[](Bitstring, Integer): Bit`

[] is used for indexing. This corresponds to Extract and Modify in SDL. Indexing starts from zero.

12.1.3 Boolean

`Boolean` is a predefined SDL-UML DataType.

It has two literals:

- `true`
- `false`

It has the following infix operations:

- `"`**`not`**`"(Boolean): Boolean`
- `"`**`and`**`"(Boolean, Boolean): Boolean`
- `"`**`or`**`"(Boolean, Boolean): Boolean`
- `"`**`xor`**`"(Boolean, Boolean): Boolean`
- `"=>"(Boolean, Boolean): Boolean`

12.1.4 Character

`Character` is a predefined SDL-UML DataType.

The syntax for literals of `Character` is defined by the lexical rule <character>. In addition, `Character` has the following literals:

NUL,	SOH,	STX,	ETX,	EOT,	ENQ,	ACK,	BEL,
BS,	HT,	LF,	VT,	FF,	CR,	SO,	SI,
DLE,	DC1,	DC2,	DC3,	DC4,	NAK,	SYN,	ETB,
CAN,	EM,	SUB,	ESC,	IS4,	IS3,	IS2,	IS1

`Character` has the following infix operations:

- `"<"(Character, Character): Boolean`
- `">"(Character, Character): Boolean`
- `"<="(Character, Character): Boolean`
- `">="(Character, Character): Boolean`

`Character` has the following operations:

- `num(Character): Integer`
- `chr(Integer): Character`

12.1.5 Duration

`Duration` is a predefined SDL-UML DataType.

The syntax for literals of the `Real` is defined by the lexical rule <real name> and there is an implicit protected operation:

- `duration(Real): Duration`

A `Real` literal used where `Duration` is required has `duration` implicitly applied.

NOTE – Because `duration` is protected, it cannot be used explicitly.

`Duration` has the following public infix operations:

- `"+"(Duration, Duration): Duration`
- `"-"(Duration): Duration`
- `"-"(Duration, Duration): Duration`
- `">"(Duration, Duration): Boolean`

- `"<"(Duration, Duration): Boolean`
- `">="(Duration, Duration): Boolean`
- `"<="(Duration, Duration): Boolean`
- `"*"(Duration, Real): Duration`
- `"*"(Real, Duration): Duration`
- `"/"(Duration, Real): Duration`

12.1.6 Integer

`Integer` is a predefined SDL-UML Datatype.

The syntax for values of `Integer` is defined by the lexical rule <integer name>.

`Integer` has the following infix operations:

- `"-" (Integer): Integer`
- `"+" (Integer, Integer): Integer`
- `"-" (Integer, Integer): Integer`
- `"*" (Integer, Integer): Integer`
- `"/" (Integer, Integer): Integer`
- `"`**`mod`**`" (Integer, Integer): Integer`
- `"`**`rem`**`" (Integer, Integer): Integer`
- `"<" (Integer, Integer): Boolean`
- `">" (Integer, Integer): Boolean`
- `"<=" (Integer, Integer): Boolean`
- `">=" (Integer, Integer): Boolean`

`Integer` has the following operations:

- `power(Integer, Integer): Integer`

12.1.7 Octet

`Octet` is a predefined SDL-UML DataType.

`Octet` is defined as a subset of `Integer`, with the constraint that an `Octet` value shall have a `length` of 8 (that is, if `oct` is an `Octet` value, `oct.length = 8` shall be true).

12.1.8 Octetstring

`Octetstring` is a predefined SDL-UML DataType.

The values of `Octetstring` are represented by the lexical rules <bit string> and <hex string>. In the case of a <bit string>, it has to have a `length` that is a multiple of 8.

`Octetstring` has the following operations:

- `bitstring(Octetstring): Bitstring`
- `octetstring(Bitstring): Octetstring`

12.1.9 Pid

`Pid` is a predefined SDL-UML DataType.

Pid is a reference to an instance of an <<ActiveClass>> <u>Class</u> (an agent in SDL).

12.1.10 Real

`Real` is a predefined SDL-UML DataType.

The syntax for literals of the `Real` is defined by the lexical rule <real name>.

`Real` has the following infix operations:

- `"-"(Real): Real`
- `"+"(Real, Real): Real`
- `"-" (Real, Real): Real`
- `"*" (Real, Real): Real`
- `"/" (Real, Real): Real`
- `"<" (Real, Real): Boolean`
- `">" (Real, Real): Boolean`
- `"<=" (Real, Real): Boolean`
- `">=" (Real, Real): Boolean`

`Real` has the following operations:

- `float(Integer): Real`
- `fix (Real): Integer`
- `power(Real, Real): Real`

12.1.11 String

`Charstring` is a predefined SDL-UML DataType defined as:

- `String : < Itemsort -> Character >`

`Charstring` represents the SDL data type `CharString`.

The syntax for values of `Charstring` type is defined by the lexical rule <character string>.

An empty `Charstring` is represented by <apostrophe><apostrophe>.

12.1.12 Time

`Time` is a predefined SDL-UML DataType.

The syntax for literals of the `Real` is defined by the lexical rule <real name> and there is an implicit protected operation:

- `time(Duration): Time`

NOTE – Because `time` is protected, it cannot be used explicitly.

A `Real` literal used where `Time` is required has `duration` and `time` implicitly applied. For example, if `t1` is a `Time` item in `t1 < 1.0` this implies `t1 < time(duration(1.0))`.

`Time` has the following public infix operations:

- `"<"(Time, Time): Boolean`
- `"<="(Time, Time): Boolean`
- `">"(Time, Time): Boolean`
- `">="(Time, Time): Boolean`
- `"+"(Time, Duration): Time`
- `"+"(Duration, Time): Time`
- `"-"(Time, Duration): Time`

- `"-"(Time, Time): Duration`

12.1.13 UnlimitedNatural

`UnlimitedNatural` is a predefined SDL-UML Datatype.

`UnlimitedNatural` is defined as a subset of `Integer`, with the constraint that an `UnlimitedNatural` value shall be >= 0.

The <asterisk> notation for an `UnlimitedNatural` is valid only in a <range> (see the syntax for <range>).

12.2 Template data types

These provide template types equivalent to SDL predefined types with context parameters.

Each of template data type is an abstract template DataType that can be bound to define a non-abstract DataType. Each abstract template DataType has the same characteristics as one of the SDL predefined data type with context parameters.

In the case of `Bag`, `Powerset`, `String`, and `Vector`, these have the same operators and expression notation as the implicit anonymous types created by using multiplicity with the type given as a parameter. However, one advantage of using a bound template type is that the data type is no longer anonymous, and therefore two items using the same name are of the same type (whereas two items each using the same type with the same multiplicity actually define two distinct – and therefore incompatible – types).

12.2.1 Attributes and parameter

There are no additional attributes, but each template has a number of parameters.

12.2.2 Constraints

- A DataType defined by a binding to a template data type shall have non-empty name.

12.2.3 Semantics

The specific semantics is defined for each specific template below.

12.2.4 Notation

The notation for a bound data type is shown using the rectangle symbol with keyword «dataType» containing the following format:

```
«dataType»
Boundtype : Templatetype < Formal -> Actual >
```

where:

`Boundtype` is the name of the data type bound to the template data type;

`Templatetype` is the name of the template data type;

`Formal` is the name of a formal parameter of the template data type; and

`Actual` is the actual parameter to replace the formal parameter.

There may be several formal and parameter pairs and these are separated by commas within the less than (<) and greater than (>) symbols, which delimit the list. For example, the `Map` (an SDL `Array)`, which has 2 parameters: the first (named `Index)` is the data type for the index value and the second (named `Itemsort`) is the data type for elements. For a data type called `CharValidity` that is `Map` of `Boolean` values indexed by `Character`, the text in the data type symbol would be:

```
CharValidity : Map < Index -> Character, Itemsort -> Boolean >
```

12.2.5 References

12.3 Array template

`Array` is a SDL-UML Datatype with two template parameters: `Index` and `ItemSort`.

`Array` maps to the `Array` of the SDL `Predefined` package.

A <composite primary> is used to apply the `Make` operator for `Array` to generate `Array` values.

A data type called `BoundMap` bound to `Array` has the following operations:

- `[](BoundMap, Index): ItemSort`

[] is used for indexing. This corresponds to Extract and Modify in SDL.

12.4 Bag template

`Bag` is a SDL-UML Datatype with one template parameter: `ItemSort`.

`Bag` maps to the `Bag` of the SDL `Predefined` package.

A <composite primary> is used to apply Make operator for `Bag` to generate `Bag` values.

An empty value of a data type bound to `Bag` is the literal `empty`.

A data type bound to `Bag` has the following methods:

- `length(): Integer` number of items in the bag

A data type called `BoundBag` bound to `Bag` has the following infix operations:

- `"`**`in`**`"(Itemsort, BoundBag): Boolean` is member of
- `"`**`and`**`"(BoundBag, BoundBag): BoundBag` intersection of bags
- `"`**`or`**`"(BoundBag, BoundBag): BoundBag` union of bags

A data type called `BoundBag` bound to `Bag` has the following operations:

- `incl(Itemsort, BoundBag): BoundBag` add item to bag
- `del(Itemsort, BoundBag): BoundBag` delete one of item from bag if present
- `"<"(BoundBag, BoundBag): Boolean` is proper subbag of
- `">"(BoundBag, BoundBag): Boolean` is proper superbag of
- `"<="(BoundBag, BoundBag): Boolean` is subbag of
- `">="(BoundBag, BoundBag): Boolean` is superbag of
- `take(BoundBag): Itemsort` removes an arbitrary item from the bag

12.5 Powerset template

`Powerset` is a SDL-UML Datatype with one template parameter: `ItemSort`.

`Powerset` maps to the `Powerset` of the SDL `Predefined` package.

A <composite primary> is used to apply Make operator for `Powerset` to generate `Powerset` values.

An empty value of a data type bound to `Powerset` is the literal `empty`.

A data type bound to `Powerset` has the following methods:

- `length(): Integer` number of items in the set

A data type called `BoundSet` bound to `Powerset` has the following infix operations:

- `"in"(Itemsort, BoundSet): Boolean` is member of
- `"and"(BoundSet, BoundSet): BoundSet` intersection of sets
- `"or"(BoundSet, BoundSet): BoundSet` union of sets

A data type called `BoundSet` bound to `Powerset` has the following operations:

- `incl(Itemsort, BoundSet): BoundSet` add item to be **in** set
- `del(Itemsort, BoundSet): BoundSet` delete item from set , so not **in** set
- `"<"(BoundSet, BoundSet): Boolean` is proper subset of
- `">"(BoundSet, BoundSet): Boolean` is proper superset of
- `"<="(BoundSet, BoundSet): Boolean` is subset of
- `">="(BoundSet, BoundSet): Boolean` is superset of
- `take(BoundSet): Itemsort` removes an arbitrary item from the set

12.6 String template

`String` is a SDL-UML Datatype with one template parameter: `ItemSort`.

`String` maps to the `String` of the SDL `Predefined` package, but the signature of `length`, `first`, and `last` use method call notation and "+" is used instead of "//" for concatenation.

A <composite primary> is used to apply Make operator for `String` to generate `String` values.

An empty value of a data type bound to `String` is the literal `emptystring`.

A data type bound to `String` has the following methods:

- `length():Integer`
- `first():ItemSort`
- `last():ItemSort`
- `append(ItemSort)`

A data type called `BoundString` bound to `String` has the following operations:

- `"+"( BoundString, BoundString ):BoundString`

  "+" infix operator. Maps to concatenation. Denoted by // in SDL
- `[](BoundString , Integer): ItemSort`

  [] is used for indexing. This corresponds to Extract and Modify in SDL.
- `substring (BoundString , Integer , Integer): BoundString`

  This correspond to substring in SDL.
- `remove (BoundString , Integer , Integer): BoundString`

  This corresponds to remove in SDL.

12.7 Vector template

`Vector` is a SDL-UML Datatype with two template parameters: `ItemSort` and `MaxIndex`.

`MaxIndex` is an `Integer` literal that specifies the maximum index value for the `Vector`.

`Map` maps to the `Vector` of the SDL `Predefined` package.

A <composite primary> is used to apply Make operator for `Vector` to generate `Vector` values.

A data type called `BoundVector` bound to `Vector` has the following operations:

- `[](BoundVector, Integer): ItemSort`

  [] is used for indexing. This corresponds to Extract and Modify in SDL.

SERIES OF ITU-T RECOMMENDATIONS

Series A	Organization of the work of ITU-T
Series D	General tariff principles
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	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
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 and next-generation networks
Series Z	**Languages and general software aspects for telecommunication systems**

Printed in Switzerland
Geneva, 2008