International Telecommunication Union

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Z.109
Amendment 1
(11/2012)

SERIES Z: LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

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

Specification and Description Language – Unified modeling language profile for SDL-2010

Amendment 1: New Appendix I – Concrete syntax

Recommendation ITU-T Z.109 (2012) – Amendment 1

ITU-T

ITU-T Z-SERIES RECOMMENDATIONS

LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

FORMAL DESCRIPTION TECHNIQUES (FDT)	
Specification and Description Language (SDL)	**Z.100–Z.109**
Application of formal description techniques	Z.110–Z.119
Message Sequence Chart (MSC)	Z.120–Z.129
User Requirements Notation (URN)	Z.150–Z.159
Testing and Test Control Notation (TTCN)	Z.160–Z.179
PROGRAMMING LANGUAGES	
CHILL: The ITU-T high level language	Z.200–Z.209
MAN-MACHINE LANGUAGE	
General principles	Z.300–Z.309
Basic syntax and dialogue procedures	Z.310–Z.319
Extended MML for visual display terminals	Z.320–Z.329
Specification of the man-machine interface	Z.330–Z.349
Data-oriented human-machine interfaces	Z.350–Z.359
Human-machine interfaces for the management of telecommunications networks	Z.360–Z.379
QUALITY	
Quality of telecommunication software	Z.400–Z.409
Quality aspects of protocol-related Recommendations	Z.450–Z.459
METHODS	
Methods for validation and testing	Z.500–Z.519
MIDDLEWARE	
Processing environment architectures	Z.600–Z.609

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

Recommendation ITU-T Z.109

Specification and Description Language – Unified modeling language profile for SDL-2010

Amendment 1

New Appendix I – Concrete syntax

Summary

Amendment 1 updates Recommendation ITU-T Z.109 to incorporate an example language definition to illustrate the application of the UML profile for SDL-2010 to a practical specification and description language.

History

Edition	Recommendation	Approval	Study Group
1.0	ITU-T Z.109	1999-11-19	10
2.0	ITU-T Z.109	2007-06-13	17
3.0	ITU-T Z.109	2012-04-29	17
3.1	ITU-T Z.109 (2012) Amd. 1	2012-11-29	17

FOREWORD

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

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

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

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

NOTE

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

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

INTELLECTUAL PROPERTY RIGHTS

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

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

© ITU 2013

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

Table of Contents

Recommendation ITU-T Z.109

Specification and Description Language – Unified modeling language profile for SDL-2010

Amendment 1

New Appendix I – Concrete syntax

1) 7.2.2 Constraints

Delete item [10], NOTE 6 (shown below), and renumber subsequent items and notes accordingly.

[10] An ownedBehavior shall not have public visibility.

NOTE 6 – A procedure or composite state type cannot be made visible outside the enclosing agent type.

2) 7.4.2 Constraints

Replace item [1] and NOTE 1 as follows:

Replace

[1] Except for <<Interface>>Interface, the general property of a Classifier shall contain at most one element.

NOTE 1 – Except for an interface definition, multiple inheritances are not allowed for SDL-2010 type definitions.

with

[1] The general property of an <<ActiveClass>> Class shall contain at most one element.

NOTE 1 – Multiple inheritance is not allowed for SDL-2010 agent type definitions.

Insert item [2] as follows and renumber subsequent items and notes.

[2] The general property of a <<DataTypeDefinition>> Class shall contain at most one supertype that has an isAbstract property of false.

NOTE 2 – Multiple inheritance is only allowed for abstract data type definitions and interface definitions (see clause 12.1.9 in [ITU-T Z.107]).

3) 8.5.2 Constraints

Replace item [2] as follows, and renumber notes accordingly:

Replace

[2] The entry and exit properties shall be empty, because entry/exit actions are not supported.

with

[2] If the submachine property is empty, the entry and exit properties shall be empty.

NOTE 1 – In SDL-2010, the definition of entry and exit procedures is only possible for composite states.

Insert item [3] as follows and renumber subsequent items and notes.

[3] If present, the entry and exit properties shall refer to a StateMachine.

4) 8.5.3 Semantics

Replace the third paragraph as follows:

Replace

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

with

The submachine property maps to *Composite-state-type-identifier*. If present, the StateMachine identified by the entry property maps to a *Procedure-definition* that defines the *Entry-procedure-definition* of a *Composite-state-graph* or a *State-aggregation-node* of the *Composite-state-type-definition* that is identified by the submachine property. If present, the StateMachine identified by the exit property maps to a *Procedure-definition* that defines the *Exit-procedure-definition* of a *Composite-state-graph* or a *State-aggregation-node* of the *Composite-state-type-definition* that is identified by the submachine property.

5) Bibliography

Add the citations in this amendment's bibliography to the bibliography in ITU-T Z.109.

6) Appendix I

Add Appendix I to ITU-T Z.109, as defined in this amendment:

Appendix I

Example language specification

(This appendix does not form an integral part of this Recommendation.)

The main body of this Recommendation defines a UML profile for SDL-2010 for the purposes of providing the semantics for formal design and specification languages. It is possible to define the abstract grammar of the formal design and specification language in terms of a UML metamodel and this language is given a precise dynamic semantics by the mapping defined by this Recommendation to the SDL abstract syntax.

The UML profile presented in this Recommendation therefore supports the use of UML as a vehicle for more formally specifying reactive systems than is usually the case with members of the UML language family.

This appendix provides an example of the concrete grammar of a practical design and specification language, and its mapping to the UML metamodel. The design and specification language illustrated in this appendix is a language for unambiguous specification and description of the way reactive systems (such as telecommunication systems, communication systems or information systems) behave. By means of the profile given in this Recommendation, specifications constructed using this practical language are formal in the sense that it is possible to analyse and interpret them unambiguously.

This appendix therefore demonstrates how this Recommendation can be utilized to define the semantics for a practical design and specification language.

The example language is presented in three parts: the Concrete grammar describes the syntax and syntactic constraints a well-formed specification shall adhere to. The Model describes transformations on the concrete syntax that are applied before the concrete syntax is mapped to the

UML metamodel. The Mapping describes the mapping of the concrete syntax to the UML metamodel defined by the UML profile given in this Recommendation. The meaning of the language is then determined as follows: the UML profile describes the mapping of the subset of the UML metamodel supported to the abstract syntax of SDL, and finally, SDL describes how to interpret a resulting specification.

While the example language represents a practical design and specification language which has been used to specify a wide range of reactive systems in application areas ranging from telecommunication network elements to enterprise information applications, the focus of this appendix is to serve as an illustration of the definition of the semantics of a practical design and specification language using the UML profile provided by this Recommendation. The techniques exhibited in this appendix can be used to define other specification languages within the UML language family. A specification language so defined can also take advantage of widely available tools based on UML metamodels.

I.1 Conventions

This clause defines the conventions used for describing the example language specification for this Recommendation. The metalanguages and conventions introduced are solely introduced for the purpose of describing the language unambiguously. The conventions of [b-ITU-T Z.111] apply throughout this document.

In the following, other than this clause, each clause may contain the following subclauses:

– The subclause entitled "Concrete grammar" specifies the concrete syntax for each non-terminal of the language.

– The subclause entitled "Model" specifies transformations on the concrete syntax that are applied before the concrete syntax is mapped to the UML metamodel.

– The subclause entitled "Mapping" provides the mapping of the concrete syntax to the UML metamodel supported in this Recommendation.

I.1.1 Concrete grammar

The Concrete grammar is specified in the format defined in [b-ITU-T Z.111], clause 5.4.2, augmented by the following additional conventions:

– Rather than using the lexical unit names <*asterisk*>, <*plus sign*>, <*vertical line*>, <*left square bracket*>, <*right square bracket*>, <*left curly bracket*> and <*right curly bracket*> to distinguish terminal symbols for characters from the symbols of the metalanguage, these terminal symbols are denoted by the corresponding characters in bold Courier New font. For example, <*left curly bracket*> and <*right curly bracket*> are denoted by **`{`** and **`}`**.

– The convention ~<*xxx*> denotes any token other than the token specified by <*xxx*>. This notation is only used in the lexical rules, see clause I.2.1.

For an element <*xxx*> of the Concrete grammar, in the text the phrase "<*xxx*> list" refers to any list of <*xxx*> items, whether possibly empty or not, or with separators or not.

To avoid cumbersome formulations, the convention is adopted that the name of the non-terminal or terminal may be prefixed by the name of the production when no confusion is likely to arise. For example, when referring to the <*identifier*> of a <*signal definition*>, instead of the cumbersome phrase "the <*identifier*> of the <*signal definition*>" the phrase "the signal <*identifier*>" may be used. Similarly, when referring to the definition that is referenced by an identifier indirectly contained in another definition, instead of the cumbersome phrase "the <*xxx*> definition referenced by the <*identifier*> of the <*name*> in <*yyy*>", it is convention to use the simpler phrase "the <*xxx*> referenced by <*yyy*>". For example, instead of "the <*class definition*> referenced by the <*identifier*> of the <*type identifier*> of the <*stimulus definition item*>" the simpler "the class referenced by the <*stimulus definition item*>" is used.

Each syntax item in this appendix is contained or used by at least one other syntax item in this appendix except:

<*specification*>

which is the container for all other items and thus the starting rule for the concrete syntax.

<*lexical unit*>

collects all lexical units.

The Concrete grammar defines the example language specification for this Recommendation and may be supplemented by a set of syntactic constraints. The syntactic constraints apply to the immediately preceding productions of the Concrete grammar. When no confusion is likely to arise, the phrase "of the <*xxx*>", where <*xxx*> refers to the preceding production of the Concrete grammar, is omitted. When a set of syntactic constraints apply to all syntax productions of a subclause, they follow the productions.

The syntax of the Concrete grammar may be further supplemented by a Model (see clause I.1.2 below). A system specification conforms to the language if it conforms to the Concrete grammar and the syntactic constraints (see [b-ITU-T Z.111], clause 5.1) after application of the Model, if present.

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

I.1.2 Model

Some constructs are considered to be shorthand notation for other equivalent concrete syntax constructs. For example, omitting an input for a signal that could possibly be received in a state is shorthand notation for an input for that signal followed by an empty transition back to the same state.

The order of application of such shorthand notations is from the outside of the specification in, unless it is specifically stated otherwise. Shorthand notations may in turn introduce syntax to which further shorthand notations can be applied. Shorthand notations are applied until further shorthand notations can no longer be applied, unless it is explicitly stated otherwise.

These transformations often create entities with unique identifiers. The phrase "an anonymous identifier" refers to such a newly created unique identifier. To avoid cumbersome formulations, the phrase "the anonymous entity", where "entity" may be a more specific kind of entity, refers to "the entity with the newly created anonymous identifier".

I.1.3 Mapping

The way a system specification behaves is defined by the mapping of the concrete syntax to a metamodel conforming to the restrictions of this Recommendation. This representation of the specification is then mapped to the abstract syntax of SDL by the mechanism defined in this Recommendation. Finally, the meaning of a conforming specification is defined by interpretation of the abstract syntax of SDL as defined in [ITU-T Z.100], [ITU-T Z.101], [ITU-T Z.102], [ITU-T Z.103], [ITU-T Z.104] and [ITU-T Z.107].

If no explicit representation is given for an alternative in a production of the textual grammar that contains a single non-terminal, then the production represents what that non-terminal represents if the production reduces to this alternative. Only those attributes and associations of the abstract syntax that are relevant for the mapping to the SDL abstract syntax are discussed. Additional attributes and associations may be populated due to constraints of UML or this Recommendation.

NOTE 1 – For example, all <*entities*> inherited by an <*agent definition*>, <*interface definition*>, <*signallist definition*> or <*class definition*> are contained in the inheritedMember property. As this property is not relevant for the mapping to the SDL abstract syntax, it is omitted in this discussion.

NOTE 2 – As a further example, there are a number of associations and intervening model elements that connect the model elements in sequential actions which do not have an impact on the mapping to the SDL abstract syntax. Consider the relation between a SendSignalAction and the destination that is its target. In the UML metamodel, the target is an ActionInputPin such that its fromAction is a ReadStructuralFeatureAction where the object is an ActionInputPin that references in its fromAction a model element representing the destination. Such intervening associations and model elements are omitted in this discussion but shall be constructed when a UML model corresponding to the Concrete grammar is created.

This Recommendation introduces stereotyped metaclasses for all metaclasses used from the UML. To simplify the presentation, when the name of the stereotyped metaclass is the same as the name of the UML metaclass, the name of the stereotyped metaclass is used. For example, instead of <<Signal>>Signal, the shorter designation Signal is used. In other words, all UML metaclasses used are considered to be implicitly stereotyped by the stereotype introduced in this Recommendation and these stereotypes are shown only when they differ from the name of the UML metaclass.

Items defined within definitions of other items are not specifically described as contained by the definition as the ownership of such contained items is apparent from the grammar. For example, an agent definition contains any agents or classes defined by enclosed agent definitions or class definitions.

I.1.4 Introductory text

Each clause or subclause may be preceded by introductory text which summarizes the purpose of the clause or subclause or otherwise introduce the definitions that follow.

This appendix may further contain paragraphs labelled as "NOTE".

I.1.5 Environment

A specified system behaves according to the stimuli exchanged with the external world. This external world is called the environment of the system being specified.

It is assumed that there are one or more computational entities in the environment, and that stimuli flowing from the environment towards the specified system have associated identities of these entities. These entities have identities that are distinguishable from any other entity identity within the specified system.

Although no assumptions can be made about the coordination and sequential order of the behaviour of the environment, the environment is assumed to obey the constraints given by the system specification.

I.1.6 Validity and errors

A system specification is a valid specification if and only if it satisfies the syntactic rules and the static conditions defined in this appendix and the abstract grammar defined by the main body of this Recommendation.

If a valid specification is interpreted and a dynamic condition is violated, then an error occurs. Predefined exceptions (see clause I.1.7) are thrown when an error is encountered during the interpretation of a system. If an exception is thrown, the subsequent behaviour of the system cannot be derived from the specification and is undetermined.

For most cases where an exception might be thrown (for example, a range check or the access of undefined variables), it is possible to include checks that test for a possible erroneous situation and perform actions that avoid behaviour that would otherwise cause an exception. Static analysis or dynamic interpretation of a specification might also indicate when it is inevitable that an exception is thrown.

I.1.7 Package Predefined

[ITU-T Z.104] defines a predefined set of data types in a package `Predefined`. The definitions in the package `Predefined` are assumed to be available in this appendix.

I.2 Lexical rules and names

This clause covers lexical units, commonly used symbols, and the visibility, resolution and use of names.

I.2.1 Lexical rules

Lexical rules define lexical units. Lexical units are terminals of the Concrete grammar.

Concrete grammar

<*lexical unit*> ::=
      <*simple name*>
    | <*quoted name*>
    | <*integer name*>
    | <*real name*>
    | <*string name*>
    | <*keyword*>
    | <*special*>
    | <*comment*>

NOTE 1 – <*simple name*>, <*quoted name*>, <*integer name*>, <*real name*> and <*string name*> are lexical alternatives for a <*name*>. The letter sequences defined by <*keyword*> items are the keywords of the Concrete grammar. The tokens defined by <*special*> represent character combinations that serve as separators, special operation identifiers or other symbols in the Concrete grammar. A <*comment*> represents comment text. Other lexical rules (such as <*letter*> or <*decimal digit*>) that are not alternatives of <*lexical unit*> are used only in the lexical rules.

The characters in a <*lexical unit*> are defined by the International Reference Version (IRV) of the International Reference Alphabet [b-ITU-T T.50]). A (non-space) control character is allowed wherever a space is allowed, and has the same meaning as a space.

IRV delete characters are ignored.

The use of the extended character set of UCS (see [b-ISO/IEC 10646]) is allowed. UCS includes IRV characters (see [b-ITU-T T.50]) as a subset. Non-printing UCS characters that do not correspond to an IRV character are treated in the same way as a control character is treated.

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

It is allowed to insert any number of spaces before or after any <*lexical unit*>. Inserted spaces or <*comment*> items have no syntactic relevance, but sometimes a space or a <*comment*> is needed to separate one <*lexical unit*> from another.

All <*lexical unit*> items except <*keyword*> items, <*uppercase letter*> items and <*lowercase letter*> items are distinct. Therefore **`AB`**, **`aB`**, **`Ab`**, and **`ab`** represent four different <*name*> items. An all-uppercase <*keyword*> is treated the same as the all lowercase <*keyword*> with the same spelling (ignoring case), but a mixed case letter sequence with the same spelling as a <*keyword*> represents a <*name*>.

For conciseness, within the lexical rules and the Concrete grammar, the lowercase <*keyword*> as a terminal denotes that the uppercase <*keyword*> with the same spelling is allowed in the same place. For example, the keyword **`default`** represents the lexical alternatives { **`default`** | **`DEFAULT`** }

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

<*simple name*> ::=
    * { <*letter*> | { <*decimal digit*>}}{_ | <*alphanumeric*>}*

<*quoted name*> ::=
    ' {<*simple name*> | <*prefix operation name*> | <*infix operation name*> |
      <*postfix operation name*>}'

If a sequence of tokens recognized as a <*name*> matches a <*keyword*>, it is not allowed as a <*name*>. For example, **`agent`** cannot be used as a <*name*>.

NOTE 2 – If a <*lexical unit*> is possibly either a <*name*> or a <*keyword*>, it is a <*keyword*>.

A <*quoted name*> resulting from attaching an apostrophe symbol `'` before and after a <*name*> may be used instead of the <*name*> wherever a <*name*> may be used.

NOTE 3 – A token representing a <*keyword*> may be used in a <*quoted name*> and may be used wherever a <*name*> may be used. For example, **`'agent'()`** may be used to call an operation **`'agent'`**, albeit it would not be allowed to use the keyword **`agent`** as name.

If two <*quoted name*> items containing a <*prefix operation name*>, <*infix operation name*> or <*postfix operation name*> item differ only in case, the semantics of the lowercase spelling applies, so that (for example) the expression **`'REM'(a, b)`** means the same as **`'rem'(a, b)`**, which means the same as **`a rem b`**.

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

<*real name*> ::=
    <*integer name*> **`.`** <*integer name*> [<*exponent*>]

<*exponent*> ::=
    {**`e`** | **`E`** } [**`-`** | **`+`**] <*integer name*>

<*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`**

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

<character string> ::=
" { <string esc> | ˜{\ | "}}* "

Control characters and spaces in a <character string> are significant: a sequence of spaces is not treated as one space in a <character string>.

<string esc> ::=
\ {b | t| n | f | r | " | \}

<hex string> ::=
′ { <decimal digit> | <hex digit> }* ′ {H| h}

<hex digit> ::=
a| b| c| d| e| f|A|B|C|D|E|F

<bit string> ::=
′ {0|1}* ′ {B|b }

<keyword> ::=
active | **agent** | **agentset** | **attribute** | **break** | **channel** | **class** | **const** |
constructor | **continue** | **create** | **default** | **do** | **else** | **entry** | **exit** |
export | **extends** | **for** | **from** | **goto** | **if** | **import** | **in** | **inout** | **input** |
interface | **literals** | **mod** | **nextstate** | **none** | **now** | **null** | **offspring** |
operation | **opt** | **out** | **output** | **parent** | **part** | **port** | **priority** | **ref** |
service | **rem** | **required** | **reset** | **return** | **save** | **self** | **sender** | **set** |
signal | **signallist** | **start** | **state** | **stereotype** | **stop** | **switch** |
syntype | **template** | **then** | **this** | **timer** | **to** | **type** | **val** | **variable** | **via** |
when | **while** | **with** | **xor** | **#else** | **#endif** | **#if** | **#ifdef** | **#ifndef**

<special> ::=
<prefix operation name>
| <infix operation name>
| <postfix operation name>
| <other special>

<prefix operation name> ::=
- | !

<infix operation name> ::=
<= | <| == | ~~ | = | >= | > | | | | - | != | !~ | // | / | ⋆ | && | + | :? | : |
mod | **rem** | **in** | **xor**

<postfix operation name> ::=
-- | ++

<other special> ::=
<<| _ | , | ; | := | :: | : | ? | /⋆ | >> | ⋆/ | .. | . | (| ({ |) | [|] | { | } | }) |@

<comment> ::=
// ˜{<line feed>| <carriage return>}* [<carriage return>] <line feed>
| /⋆ { ˜{⋆/}| <comment> }* ⋆/

The tokens <line feed> and <carriage return> refer to the characters at position 0/10 and 0/13, respectively, of the IRV C0 set.

Control characters and spaces in a <*comment*> are significant: a sequence of spaces is not treated as one space in a <*comment*>.

NOTE 4 – Comments bounded by `/*` and `*/` may contain nested comments.

I.2.2 Comment stereotype

A special stereotype is available to indicate comments that are guaranteed not to be treated as space, but instead are considered part of the specification.

Concrete grammar

The comment stereotype is indicated by a <*stereotype item*> that is a <*named value*> with the <*name*> **`comment`**.

NOTE 1 – The <*expression*> in the comment stereotype is the text of the comment and is typically a <*character string*>.

Mapping

NOTE 2 – The mapping for a comment stereotype is given in clause I.4.5.

I.2.3 Namespace and visibility of names

A namespace defines a context within which a name is valid and can be used. When a name is valid and can be used, it is called "visible". Each name belongs to a namespace, which is the namespace enclosing the definition of the name.

The following productions introduce namespaces: <*package definition*>, <*class definition*>, <*agent definition*> and <*interface definition*>.

Concrete grammar

A namespace may allow a <*name*> to be visible outside of the namespace. The specification of visibility controls the visibility of the <*name*> outside of a namespace. Visibility is specified by the application of the stereotypes <<`public`>>, <<`protected`>> and <<`private`>>. A <*name*> can be used in an <*identifier*> to reference an <*entity*> defined in a namespace from outside of that namespace only if that <*name*> is visible outside of the namespace.

An <*import definition*> introduces <*name*> items from another package into the namespace of the containing <*definition unit*>, see clause I.3.3, so that these <*name*> items can be used in an <*identifier*> without a <*qualifier list*>.

To use an <*identifier*> to reference an <*entity*>, the <*name*> of the entity has to be visible. Namespaces may restrict the visibility of the <*name*> of an <*entity*> defined within the namespace. Restricting visibility avoids every <*name*> having global visibility, so that the same <*name*> can be used in different contexts for different entities.

Namespaces are scope units. In addition, a namespace may contain scope units that may introduce local variable <*name*> items which hide more global <*name*> items visible in a namespace from view within that scope unit. Scope units are defined by the following non-terminal symbols of the concrete grammar: <*package definition*>, <*class definition*>, <*agent definition*>, <*interface definition*>, <*signallist definition*>, <*service definition*>, <*state definition*>, <*operation definition*>, <*constructor definition*>, <*entry action*>, <*exit action*>, <*start transition*>, <*input*>, <*continuous transition*>, <*spontaneous transition*>, <*labeled transition*>, <*connect transition*>, <*compound statement*>, <*if statement*>, <*decision statement*>, <*type decision statement*>, <*for statement*>, <*while statement*> and <*compound expression*>.

The following productions may introduce local variable <*name*> items into a scope unit: <*class definition*>, <*agent definition*>, <*service definition*>, <*state definition*>, <*stimulus*>, <*statements*>, <*loop clause*> and <*compound expression*>. Local variable <*name*> items cannot be made visible outside their scope unit and cannot have a visibility stereotype applied.

Model

If an <*entity*> does not contain a visibility stereotype, this is shorthand notation for having default visibility applied, as described in Table I.2.1.

Table I.2.1 – Default visibility of entities within a given context

Context	Entity	Default visibility
Package	all entities	public
Class	operation realizing interface	public
Class, agent	constructor	public
Class, agent	all other operations	protected
Class, agent	attribute	private
Agent	port	public
Agent	channel	protected
Agent	signal, signallist	protected
Class	literal	public
Class, agent	interface, class, agent, syntype	protected
Interface	operation	public

Mapping

A name represents a NamedElement. If the stereotype <<`public`>>, <<`private`>> or <<`protected`>> is applied, visibility is public, private or protected, respectively.

I.2.4 Identifiers, names and name resolution

A specification consists of a hierarchy of definitions that each associate a name with an entity. There are various different kinds of entities, including packages, classes or operations. In addition, name items may reference defined entities or may reference properties of entities within the context of an entity.

A name is established by the definition of an entity. The following productions are definitions: <*package definition*>, <*export definition*>, <*renaming*>, <*class definition*>, <*constructor definition*>, <*agent definition*>, <*interface definition*>, <*signallist definition*>, <*signal definition*>, <*timer definition*>, <*syntype definition*>, <*port definition*>, <*channel definition*>, <*attribute definition*>, <*variable definition*>, <*literal definition*>, <*agentset definition*>, <*operation definition*>, <*parameter*>, <*result*>, <*service definition*>, <*state definition*>, <*state connection points*>, <*state list*>, <*labeled transition*>, <*labeled statement*>, <*loop variable definition*> and <*context parameter*>.

It is allowed to use same name for different entities, as long as they belong to different entity kind groups or are defined in different namespaces, but the identity of the entity includes the context of the definition, its kind, and possibly other properties such as the signature in the case of an operation. The definition context is the path to the definition of an entity from an outer level package or the system.

The identity of an element used in the specification (other than in its definition) is usually established through an identifier, which includes a name and may include a qualifier list that gives the path to the definition of the entity. However, if the complete qualifier list were given in the concrete syntax for every identifier, these qualifier lists would obscure the specification and make it tedious to write. For this reason it is allowed to omit the qualifier list or part of the qualifier list in the concrete syntax of an identifier if the element is unambiguously established.

Concrete grammar

<*identifier*> ::=
    <*qualified name*> | <*unqualified name*>

<*qualified name*> ::=
    <*qualifier list*> <*name*>

<*unqualified name*> ::=
    <*name*>

<*qualifier list*> ::=
    <*absolute qualifier list*> | <*relative qualifier list*>

<*absolute qualifier list*> ::=
    :: <*qualifier*>*

An <*absolute qualifier list*> gives the full path from the system specification to the entity definition.

<*relative qualifier list*> ::=
    <*qualifier*>*

If the <*qualifier list*> is omitted or if a <*relative qualifier list*> is used (i.e., the qualifier list does not give the full path to the entity definition referenced by <*name*> in <*base identifier*>), the full path is determined by name resolution, see below.

<*qualifier*> ::=
    <*name*> [<*actual context parameters*>] **::**

The <*name*> in <*qualifier*> shall reference a type or a package.

Either the <*qualifier*> refers to a supertype of an entity defined by a type definition or the <*qualifier*> reflects the logical hierarchical structure from the system level to the defining context, such that the system level is the leftmost textual part. The <*identifier*> of an entity is then represented by the qualifier, the <*name*> of the entity, and, only for operations, the signature.

<*name*> ::=
    <*simple name*>
    | <*quoted name*>
    | <*integer name*>
    | <*real name*>
    | <*string name*>

The <*name*> of an <*entity*> which is referenced by an <*identifier*> is determined by name resolution as given below. Name resolution identifies the definition for an <*identifier*>; it selects the definition of the <*entity*> among all the visible alternative definitions, if any.

Except for a <*name*> in an <*operation definition*>, each distinct <*name*> in a specification always corresponds to a distinct token and each occurrence of the <*name*> corresponds to the same token. In the case of a <*name*> in an <*operation definition*>, the token depends also on the signature of the operation (the parameters and the result).

A scope unit may contain a list of entity definitions. Each of the definitions defines one or more <*entity*> items belonging to a certain entity kind and having an associated <*name*>. Each <*entity*> has its defining context in the scope unit that defines it.

Entities are grouped into entity kinds. The following entity kind groups exist:

a) package;

b) agent;

c) class, interface;

d) channel, port;

e) signal, timer, signallist;

f) variable, parameter, attribute, agentset, state;

g) literal, operation;

h) service;

i) label.

Each entity of a kind group shall have an <*identifier*> different from any other entity of the same kind group. A syntype (see clause I.9.1.7) belongs to the kind group its <*parent type*> belongs to. If a <*name*> is introduced in a <*renaming*> (see clause I.3.3) it belongs to the kind group the <*identifier*> of the <*renaming*> belongs to.

NOTE 1 – Consequently, no two definitions in the same scope unit and belonging to the same entity kind group shall have the same <*name*>, except operations defined in the same <*class definition*> that differ in at least one argument <*type*>.

Entities with the same <*name*> are allowed in a scope if they are of different kind groups. For example, the same <*name*> can be used for an agent and a class, but the same <*name*> cannot be used for a class and an interface.

Operations with the same <*name*> are allowed in the same scope if they have different signatures, see clause I.9.1.5.

Entities of the same kind and with the same <*name*> (and for operations, with the same signature) shall not appear as entities in a type and in a supertype unless they are virtual, see clause I.4.3.

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

If a <*qualifier*> can be understood both as qualifying by an enclosing scope and as qualifying by a supertype, it denotes an enclosing scope.

An entity can be referenced using an <*identifier*>, if the <*identifier*> can be bound to the entity definition using resolution by container or, for entities of entity kind group g), resolution by context.

The binding of an <*identifier*> to a definition through resolution by container proceeds in the following steps, considering every entity kind valid for the context where the <*identifier*> occurs, and starting with the scope in which the <*identifier*> occurs, if the <*identifier*> has a <*relative qualifier list*> or no <*qualifier list*> is present; or starting with the system scope, if the scope unit has an <*absolute qualifier list*>; or starting with the scope defined by the (static) type of the <*target*>, if the <*identifier*> occurs in an <*invocation*>:

a) If a visible <*package definition*> or type definition exists with the same <*name*> as the leftmost <*qualifier*>, resolution by container for the <*identifier*> with the leftmost <*qualifier*> removed is attempted in that <*package definition*> or type definition; otherwise

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

c) if the <*identifier*> is an <*invocation*>, resolution by container is performed in the scope unit defined by the static type of the <*target*>; otherwise

d) if the scope unit is a specialized type definition, resolution by container is performed in the scope unit defined by a <*super type*> until the <*identifier*> is bound to a unique entity that is a visible constructor or literal, or a type is reached that has no specialization or instantiation; otherwise

e) if the scope unit is a <*template instantiation*>, resolution by container is performed in the scope unit defined by the <*base identifier*> until the <*identifier*> is bound to a unique entity that is a visible constructor or literal, or a type is reached that has no specialization or instantiation; otherwise

f) if the scope unit has an <*import definition*> and a unique entity exists with the same <*identifier*> and a valid entity kind and is visible in the referenced <*package definition*>, the <*identifier*> is bound to that entity; otherwise

g) if the scope unit has an <*interface definition*> or <*signallist definition*> and a unique entity exists with the same <*identifier*> and a valid entity kind and is visible in the <*interface definition*> or <*signallist definition*>, respectively, the <*identifier*> is bound to that entity; otherwise

h) resolution by container is performed in the scope unit that defines the current scope unit unless that scope unit is a <*package definition*>.

NOTE 2 – In other words, <*name*> items reference entities that are declared in the current scope, or in an enclosing scope (unless the current scope is a <*package definition*>), or have been imported into the current scope. A <*qualified name*> is resolved by resolving the <*qualifier list*> and then resolving the <*identifier*> in the namespace referenced by the <*qualifier list*>. When the <*qualifier list*> contains multiple <*qualifier*> items, the resolution is iterative, starting from the leftmost qualifier, or the system specification, if an <*absolute qualifier list*> is used.

When an <*identifier*> references an entity that belongs to entity kind group g), the binding of the <*identifier*> to a definition shall be resolvable by context. Resolution by context is attempted after resolution by container; that is, if binding of an <*identifier*> through resolution by container is possible and this binding does not violate any static constraints, this binding is used even if resolution by context could bind that <*identifier*> to another entity also.

NOTE 3 – Consequently, resolution by context is only applied if a unique binding that satisfies all static constraints cannot be found through resolution by container.

The context for resolving an <*identifier*> is an <*assignment*> (if the <*identifier*> occurred in an <*assignment*>), a <*decision statement*> (if the <*identifier*> occurred in the <*expression*> or <*constraint*> of a <*decision statement*>), a <*type decision statement*> (if the <*identifier*> occurred in the <*expression*> or <*type*> items of a <*type decision statement*>) or an <*expression*> that is not part of any other <*expression*>, otherwise. Resolution by context proceeds as follows:

a) For each <*identifier*> occurring in the context, find the set of visible <*name*> items matching the <*name*> of <*identifier*> and a valid entity type for the context.

b) Consider only those elements that do not violate any static type constraints. Each remaining element represents a possible, statically correct binding of the <*identifier*> items in the <*expression*> to entities.

c) Remove all elements that represent an operation definition in a supertype of a type, where an operation definition with equal signature is present in the type, or where an operation is present in the type such that the signature of this operation can be converted into the signature of the operation in the supertype by applying the conversions below, or vice versa.

d) For each identified operation, determine the conversions that have to be applied to the actual parameters to match the parameters in the identified operation definition, if any. For each operation, count the number of mismatches considering the worst conversion (see below) in any argument.

e) Compare the elements in pairs, dropping those with more mismatches, if both elements in a pair have the same worst conversion applied, or dropping those with the worse conversion (see below).

f) If there is more than one remaining element, all non-unique <*name*> items shall represent the same operation signature; otherwise in the context it is not possible to bind the <*identifier*> items to a definition.

The following conversions are applicable to an expression in step d) of resolution by context:

1. coercion of the dynamic type of the expression to a supertype, if the expression corresponds to an <*in parameter*>; or
2. substitution of the default for an omitted parameter; or
3. if the number of actual parameters is more than the number of parameters and the last parameter is a list type, and all actual parameters including the actual parameter corresponding by position to the last parameter have a type compatible with the <*type identifier*> in the list type, replacement of those actual parameters by a single expression returning a collection of those actual parameter expressions (see clause I.9.2.6).

In steps d) and e) of resolution by context, a conversion further down the above list is worse than a conversion higher up in this list.

A class may realize operations with the same <*name*> and signature from two separate interfaces. An agent may use signals with the same <*name*> from two separate signallists. In these situations, the <*name*> items are considered to be references to the same definition and can be used without qualification.

When an <*identifier*> references an entity that does not belong to entity kind group g), the <*identifier*> is bound to an entity that has its defining context in the nearest enclosing scope unit in which the <*qualifier list*> of the <*identifier*> is the same as the rightmost part of the <*absolute qualifier list*> denoting this scope unit. If the <*identifier*> does not contain a <*qualifier*>, then the requirement on matching of <*qualifier*> items does not apply.

Mapping

If the <*qualifier*> list is empty, a <*qualified name*> references a <name> defined in or imported into the current namespace. If the <*qualifier*> list is not empty, the <*qualified name*> references a <name> in another namespace. The <*name*> represents the <u>name</u> of the defined or imported element.

NOTE 4 – The interpretation of a <*template instantiation*> is given in clause I.4.4.1.

A <*qualifier*> indicates the Namespace a name is defined in. The <*identifier*> represents the <u>name</u> in the Namespace.

NOTE 5 – The interpretation of a qualifier with <*actual context parameters*> is given in clause I.4.4.1.

I.3 Organization

It is not usually practical to describe a system in a single definition unit. The language therefore supports the partitioning of the specification into a number of definition units and the use of packages to organize the specification.

I.3.1 Entities

Each entity is defined within a namespace, which is the entity or package within which it is defined. Entities in the system specification are contained in an unnamed global namespace.

Table I.3.1 lists the kinds of entities that may appear in a specification, where (that is, at what level of a specification) they may appear, and whether their definition may be parameterized. "Global" refers to the outermost namespace, that is, the unnamed namespace introduced by the system specification (the term "definition" is omitted from this table for conciseness).

Concrete grammar

<entities> ::=
  { <stereotype>* <entity>* }
  | **;**

<entity> ::=
  <import definition>
  | *<export definition>*
  | *<visibility clause>*
  | *<class definition>*
  | *<agent definition>*
  | *<interface definition>*
  | *<signallist definition>*
  | *<syntype definition>*
  | *<signal definition>*
  | *<timer definition>*
  | *<attribute definition>*
  | *<variable definition>*
  | *<agentset definition>*
  | *<service definition>*
  | *<state definition>*
  | *<transition>*
  | *<start transition>*
  | *<labeled transition>*
  | *<entry action>*
  | *<exit action>*
  | *<operation definition>*
  | *<constructor definition>*
  | *<literal definition>*
  | *<channel definition>*
  | *<port definition>*
  | *<selected entities>*

Mapping

Each <*entity*> in the <*entity*> list represents the definition of model elements contained in the containing definition.

Each <*entity*> represents the definition of a model element. The containing definition represents <u>owner</u>.

Table I.3.1 – Entity kinds

Entity	May appear in	May be specialized	May be parameterized
definition unit	global, package	No	No
package	global, package	No	No
export	package	No	No
import	global, package	No	No
visibility clause	package	No	No
class	global, package, agent, class	Yes	Yes
agent	global, package, agent	Yes	Yes
interface	global, package, class, agent	Yes	Yes
signallist	global, package, agent	Yes	Yes
syntype	global, package, agent, class	Yes	No
signal	global, package, agent, signallist	No	Yes
timer	global, package, agent	No	No
attribute	class	No	No
variable	global, package, agent, class, service, statements	No	No
agentset	agent	No	No
service	global, package, agent	No	No
state	service	No	No
transition	service	No	No
start transition	service	No	No
labeled transition	service	No	No
entry action	composite state	No	No
exit action	composite state	No	No
operation	global, package, class, agent, service	No	Yes
constructor	class, agent, service	No	Yes
literal	class, agent	No	No
channel	agent	No	No
port	agent	No	No
selected entities	any entity list	No	No

I.3.2 Specification

A <*specification*> is described as a <*system specification*>, possibly augmented by a set of <*package definition*> items. A <*package definition*> allows definitions to be used in different contexts by referencing the definitions in the package in these contexts or by importing <*name*> items from the package into these contexts, see clause I.3.3.

Concrete grammar

<*specification*> ::=
    <*package definition*>* <*system specification*>

<*system specification*> ::=
    <*definition unit*>* <*system*>

<*definition unit*> ::=
    <*entity*>*

A <*definition unit*> corresponds to a file which contains a set of <*entity*> items.

NOTE 1 – The manner in which the <*system*> is provided is not defined in this appendix.

An <*entity*> in a <*definition unit*> shall be one of the following: <*class definition*>, <*agent definition*>, <*interface definition*>, <*signallist definition*>, <*syntype definition*>, <*service definition*>, <*operation definition*>, <*signal definition*>, <*timer definition*>, <*variable definition*>, <*import definition*>, <*export definition*> <*visibility clause*>.

A <*variable definition*> in a <*definition unit*> shall have <*immutability*> `const` defined.

NOTE 2 – Variables in definition units are global to the (system or package) context and cannot be modified.

Model

If the definition of the package `Predefined` is not present in a <*specification*>, the package definition of package `Predefined` is inserted into the <*package definition*> list of the <*specification*>.

I.3.3 Package

In order for definitions to be used in different systems they have to be defined within a package.

There is no special syntax to define packages; packages are represented by directories. A directory may contain a number of files. A file represents a <*definition unit*> and may contain multiple entity definitions.

A top-level directory represents a global namespace. This directory may contain directories (representing packages) or files (representing definition units). Each directory, representing a package, may in turn contain directories (representing packages) or files (representing definition units).

A package may export a list of name items or it may export a list of name items associated with a name that could be used to reference that list of name items in a subsequent import.

A definition unit may import a list of name items from another package or from a type within the current package after which these name items can be used without a qualifier list to refer to the definitions in the original package or in the type.

Concrete grammar

<*package definition*> ::=
    <*definition unit*>* <*package definition*>*

A <*package definition*> corresponds to a directory of the file system. The <*name*> of the package is the name of the directory.

Each file in a directory corresponding to a package definition corresponds to a <*definition unit*> (see clause I.3.2). All <*entity*> items in a <*definition unit*> of a package belong to the package.

Each directory in a directory corresponding to a package definition corresponds to a contained <*package definition*>.

If the <*system specification*> is omitted in a <*specification*>, there shall exist a mechanism for using the <*package definition*> items in other <*specification*> items. The mechanism is not otherwise defined in this appendix.

<*export definition*> ::=
    `export` { <*named export*> | <*unnamed export*> | <*asterisk export*> | <*no export*> }**`;`**

If <*no export*> is present in an <*export definition*>, no other <*export definition*> items shall be present in the containing <*package definition*>.

<*named export*> ::=
    <*name*> **=** <*exported names*>

<*unnamed export*> ::=
    <*exported names*>

<*asterisk export*> ::=
    ⋆

<*no export*> ::=
    `none`

<*exported names*> ::=
    <*exported name*>+[| **,**]

<*exported name*> ::=
    [<*kind*>] <*identifier*>

The <*identifier*> in an <*exported name*> shall reference a visible and public <*name*>. The <*qualifier*> list of the <*identifier*> items in an <*exported name*>, if present, shall contain at most one <*qualifier*> which shall reference a type definition in the package containing the <*export definition*>. If such <*qualifier*> is present, the <*name*> shall not be the <*name*> of a type definition.

An <*exported name*> may contain a <*name*> in its <*identifier*> which the package has imported from other packages.

<*import definition*> ::=
    `import` { <*named import*> | <*unnamed import*> | <*asterisk import*> | <*local import*> }**;**

<*named import*> ::=
    <*imported names*> **`from`** <*identifier*>

<*unnamed import*> ::=
    <*identifier*>

<*asterisk import*> ::=
    ⋆ **`from`** <*identifier*>

<*local import*> ::=
    <*imported names*> **`from`** <*name*>

The <*identifier*> in a <*named import*>, <*unnamed import*> and <*asterisk import*> shall reference a visible <*package definition*>. This package shall be part of the <*specification*> or a package contained in another package or else there shall exist a mechanism (not defined by this appendix) for accessing the referenced <*package definition*>, just as if it were a part of the <*specification*>. All <*package definition*> items referenced by the <*name*> items in the <*qualifier*> of the <*identifier*>, if present, shall be visible relative to the containing package. A package is visible if its <*identifier*> is visible according to the visibility rules for <*identifier*> items. The visibility rules imply that a package is visible in the package in which it is logically contained.

The <*name*> in a <*local import*> shall reference a <*class definition*> or <*agent definition*> contained in the current package or be an imported <*name*> referencing a <*class definition*> or <*agent definition*>.

An <*import definition*> shall be used only when no name collision occurs. A name collision occurs when the import definition introduces a name into the namespace of a package where a name of the same kind is already present in that namespace.

<*imported names*> ::=
    <*imported name*>+[| `,`]

<*imported name*> ::=
    [<*kind*>] { <*identifier*> | <*renaming*> }

The <*identifier*> in an <*imported name*> of a <*named import*> shall reference a <*name*> visible in the package referenced in the containing <*named import*>.

The <*identifier*> in an <*imported name*> of a <*local import*> shall reference a visible <*name*>.

The <*qualifier*> list of the <*identifier*> items in an <*imported name*>, if present, shall contain at most one <*qualifier*> which shall reference a type definition in the package referenced in the <*import definition*>.

<*renaming*> ::=
    <*name*> `=` <*identifier*>

<*kind*> ::=
    `operation` | **`service`** | **`state`** | **`signal`** | **`timer`** | **`agent`** | **`class`** | **`interface`** |
    `signallist` | **`type`** | **`literals`** | **`syntype`** | **`channel`** | **`port`** | **`attribute`** |
    `variable` | **`agentset`** | **`package`**

The <*kind*> **`type`** is used for selection of the <*name*> of a type definition and also of a syntype <*name*>.

The <*kind*> in an <*exported name*> or <*imported name*> denotes the entity kind of the <*name*> exported or imported, respectively. Any pair of <*kind*> and <*identifier*> shall be distinct within an <*import definition*>. For an <*imported name*>, <*kind*> shall be omitted if and only if the <*name*> of the <*identifier*> is unique in the package referenced by the <*import definition*>.

<*visibility clause*> ::=
    <*identifier*>+[| `,`] <*stereotype*>* `;`

A <*visibility clause*> shall not contain the stereotype <<protected>>.

A <*visibility clause*> shall not contain a <*stereotype*> that relaxes the visibility constraints imposed by the referenced definition.

Model

Each package shall import implicitly or explicitly the package **`Predefined`**. Each package shall implicitly or explicitly import all public <*name*> items from the package **`Predefined`**, including the public <*name*> items for all operations of classes defined in the package **`Predefined`**. Every package shall include an <*unnamed import*> of the package **`Predefined`**, if equivalent <*import definition*> items are not already present in that package. Further, every package shall include a <*local import*> listing all public operator <*name*> items for each <*class definition*> imported from the package **`Predefined`** in its <*imported names*>, if such <*local import*> is not already present in that package.

If a package does not contain an <*export definition*>, this is shorthand for an <*asterisk export*>.

When an <*export definition*> contains <*asterisk export*>, this is shorthand for an <*export definition*> that lists all public <*name*> items defined in the package and all public <*name*> items that are imported from other packages in the <*exported names*> of its <*unnamed export*>. The <*exported name*> items also contain the <*kind*> of each exported <*name*>.

If an <*import definition*> contains a <*name*> in the <*imported names*> of a <*named import*> where that <*name*> is the <*name*> of a <*named export*> in an <*export definition*> of the <*package definition*> referenced by the <*identifier*>, then the <*name*> is replaced by the list of <*name*> items contained in the <*exported names*> of the <*named export*>. After this transformation, the referenced

<*named export*> is transformed into an <*unnamed export*> with the <*exported names*> as its <*exported names*>.

An <*unnamed import*> is shorthand for a <*named import*> such that every <*name*> of an <*identifier*> in any <*exported name*> item of the <*package definition*> referenced by the <*identifier*> of the <*unnamed import*> appears as an <*imported name*> of the <*named import*>. If the <*exported name*> contains <*kind*>, the <*imported name*> contains that <*kind*> also.

When an <*import definition*> contains <*asterisk import*>, this is shorthand for an <*import definition*> that lists all public <*name*> items in the package referenced in the <*identifier*> and all public <*name*> items imported into the referenced package in the <*imported names*> of its <*named import*>.

For every <*name*> of an <*imported name*>, a <*visibility clause*> is created with the stereotype <<private>>.

An element shall have public visibility to be able to be exported from a namespace or imported into a namespace. If a <*visibility clause*> is present where a <*qualified name*> in the <*qualified name*> list is the name of an element, the visibility of the element is considered to be as defined in the <*stereotype*> items. The visibility of the element within the package is not affected by the <*visibility clause*>.

Mapping

A <*package definition*> represents a Package. A name represents name. The <*stereotype*> items in the <*stereotype*> list apply to the Package. Each <*entity*> in <*entities*> represents an ownedMember. A package uses another package if either the fully qualified name of the used package is the <*qualifier*> of a name referenced in the using package or if the using package contains an <*import definition*> where <*qualified name*> is the fully qualified name of the used package.

An <*export definition*> establishes the exported elements of the containing package.

NOTE – An <*export definition*> does not represent a model element. The exported elements are used to establish what an <*import definition*> represents.

The package exports all elements identified in its <*exported names*>.

Each <*imported name*> in the <*imported names*> of an <*import definition*> represents ElementImport. The importedElement is represented by the <*qualified name*>. If <*kind*> is present, the imported element shall be of the indicated kind. If <*renaming*> is present, the <*name*> represents the alias, and the <*qualified name*> represents the importedElement. The current namespace is the importingNamespace and visibility is private. The package the element is imported from is determined based on the <*qualifier*> list. If an <*absolute qualifier list*> is used, the element is imported from the package identified by following the package name items constituting the <*qualifier*> list from the root of the package hierarchy; otherwise, the element is imported from the package identified by following the package name items from the current package.

I.4 Basic concepts

This clause introduces language mechanisms to support the modeling of application-specific phenomena by instances and application-specific concepts by types. The concepts of type and instance and their relationship are fundamental to a specification. This clause introduces the basic semantics of type definitions, templated definitions, template instantiations, binding of context parameters, specialization and instantiation.

I.4.1 Types and instances

Type definitions introduce named entities, referred to as types, that define the behaviour and structure of a set of data items, referred to as the instances of the type. A data item always has a type which defines its behaviour and its structure.

A type describes a set of properties. All instances of the type have this set of properties. An example of a type definition is an agent definition. An example of a property of this type is a service definition. An example of a set of properties is an agentset definition.

The following productions are type definitions: <*class definition*>, <*agent definition*>, <*interface definition*> and <*syntype definition*>.

The following productions represent definitions of properties of types: <*agent definition*>, <*class definition*>, <*interface definition*>, <*syntype definition*>, <*signal definition*>, <*service definition*>, <*state definition*>, <*entry action*>, <*exit action*>, <*start transition*>, <*labeled transition*>, <*input*>, <*save*>, <*spontaneous transition*>, <*continuous transition*>, <*connect transition*>, <*channel definition*>, <*port definition*>, <*signallist definition*>, <*operation definition*>, <*constructor definition*>, <*literal definition*>, <*attribute definition*> and <*agentset definition*>.

Clause I.5.1 introduces type definitions for agents, while other type definitions are introduced in clause I.9.1.1 (class) and clause I.9.1.2 (interface).

An instance is created by the instantiation of a type. An example of an instance is an agent instance, which is an instantiation of an agent. An instance of a particular type has all the properties defined for that type. A type may be declared as abstract, in which case the type shall not be instantiated.

Specialization allows one type, the subtype, to be based on another type, its supertype. A subtype inherits all the properties of the supertype. The subtype may further add properties to those inherited from the supertype or it may redefine virtual properties of the supertype. A virtual property is allowed to be constrained in order to provide for analysis of the more general types.

A parameterized (templated) type is a type where some entities are represented as context parameters. A context parameter of a type definition may have a constraint, limiting the actual parameters that can be bound (see clause I.4.4.1). The constraints allow static analysis of the parameterized type. An example of a parameterized type is a parameterized agent definition where one of its contained agentsets is specified by a type context parameter; this allows the parameter to be of different types in different contexts, that is, instances of that agent may contain agentsets of different agents depending on the context.

The type of an instance (or instance set) based on a parameterized identifier is the anonymous type formed by binding the parameters of the parameterized identifier to the actual parameters given. Binding all context parameters of a parameterized type yields an unparameterized type. There is no subtype relationship between a parameterized type and the type derived from it.

Concrete grammar

<*type*> ::=
    <*type identifier*>
    | <*constrained type*>
    | <*list type*>

<*type identifier*> ::=
    <*identifier*> | <*template instantiation*> | **`stop`**

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

NOTE 1 – To avoid cumbersome text, this appendix relies on the convention that the phrase "the type S" (or "the S type") is used instead of "the type defined by the type definition with <*name*> S in its <*type identifier*>", when no confusion is likely to arise.

NOTE 2 – The keyword **`stop`** represents the <*type identifier*> of the stop signal, see clause I.7.2.

<*constrained type*> ::=
    <*type identifier*> <*constraint*>

The <*expression*> items in each <*range*> of the <*constraint*> shall be constants.

<*list type*> ::=
  { <*type identifier*> | <*constrained type*> }⋆

A <*list type*> shall be present only in the <*parameters*> of a <*constructor definition*> or an <*operation definition*> and in an <*operation context parameter*>.

Model

A <*list type*> is transformed into a <*syntype definition*> with an anonymous <*name*> and a parent type that is an anonymous type derived from the predefined type **`String`** with the <*type identifier*> in the <*list type*> as type parameter.

A <*constrained type*> is shorthand notation for an implied <*syntype definition*> having an anonymous <*name*>. This anonymous <*syntype definition*> is constructed from the <*constrained type*> by using <*type*> as the <*type*> and <*constraint*> as the <*constraint*>.

Mapping

A <*type*> that is a <*type identifier*> references a DataTypeDefinition or an <<`ActiveClass`>>Class. The <*type identifier*> represents the name of the definition. A <*constrained type*> represents a <<`Syntype`>>Class where the <*type identifier*> in <*type*> represents name and the <*constraint*> represents constraint. The interpretation of <*list type*> is given in the Model above.

I.4.2 Abstract type

A type is abstract if its definition contains the stereotype <<`abstract`>> or has unbound context parameters.

Concrete grammar

The stereotype <<`abstract`>> specifies that the entity this stereotype is attached to is abstract; it applies to <*class definition*>, <*agent definition*> and <*operation definition*>.

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

A signal defined in <*signal definition*> referencing an abstract <*class definition*> shall not be referenced in an <*output*>. The constructor for a class defined by an abstract <*class definition*> shall not appear in an <*operation call*>. The constructor for an agent defined by an abstract <*agent definition*> shall not appear in a <*create request*>.

I.4.3 Specialization

A type definition may specify a type as a specialization of another type (the supertype), yielding a new subtype. A subtype has all the properties of the supertype and optionally has properties in addition to the properties of the supertype. The subtype optionally redefines virtual properties of the supertype.

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

I.4.3.1 Properties of specialized types

A specialized type is based on a type, or a set of types (its supertypes) and defines a new type separate from its supertypes. The entities of a specialized type definition consists of the entities of the supertypes and the entities added by the specialization.

Concrete grammar

<*specialization*> ::=
  `:` <*super type*>+[| **`,`**]

<*super type*> ::=
    <*type identifier*>

At most one of the <*super type*> items in the <*super type*> list shall reference a type definition other than an <*interface definition*>.

NOTE – In other words, multiple inheritance is allowed only for interfaces.

A <*type identifier*> in a <*super type*> list references the definition of a supertype of the specialized type. The specialized type is said to be a subtype of the supertype. Any specialization of the subtype is also a subtype of the supertype.

If a type is (directly or indirectly) a subtype of another type, the supertype, then:

a) the definition of the supertype shall not contain the definition of the subtype;

b) the definition of the supertype shall not be a specialization of the subtype;

c) definitions contained by the definition of the supertype shall not be specializations of the subtype.

A specialized type definition shall not contain a property with the same <*name*> (and in the case of operations, also with the same <*operation signature*>) for a property defined in a supertype, unless the property in the subtype redefines the property in the supertype.

An inherited property of a type may be redefined in a subtype of the type if the property is virtual. An inherited property of a type shall not be redefined in a subtype of the type if the property is not virtual.

Mapping

The interpretation of <*specialization*> is given in the clauses where <*specialization*> is used.

I.4.3.2 Virtuality and redefinition

Within a type, its properties (see clause I.4.1) may be given a virtuality indicating that when the containing type is specialized it is allowed to redefine these properties in the specialization.

The impact of being virtual on a property differs with the property. For example, virtual operations are invoked (dispatched) based on the dynamic types of their target. Virtual start transitions replace the existing start transition invoked when the agent is created.

Concrete grammar

Virtuality applies to all <*entity*> items representing properties of types and is expressed by the stereotypes <<`virtual`>>, <<`redefined`>> or <<`finalized`>>.

When the stereotype <<`virtual`>> or <<`redefined`>> is applied to a property, the property is a virtual property. A virtual property may be redefined in a specialization of the enclosing type.

When the stereotype <<`final`>> is applied to a property, the property is not a virtual property. If a property is not virtual, the property shall not be redefined in a specialization of the enclosing type.

A redefined property is a property having <<`redefined`>> or <<`final`>> as virtuality. Every redefined property shall be directly or indirectly (via another redefined property) a redefinition of a virtual type that is not redefined.

A virtual property that is a type is a virtual type. A virtual type may have an associated virtuality constraint which is an <*identifier*> referencing a definition of the same entity kind as the specialized type. The virtuality constraint is the <*expression*> of a <*named value*> in the virtuality stereotype.

If a virtuality constraint is present and does not reference the type of the virtual property being constrained, the <*type identifier*> in <*specialization*>, if present, shall be the same type or a subtype of the type referenced by the virtuality constraint.

A virtual property and its constraint shall not have context parameters.

Accessing a virtual type by means of a qualifier denoting one of the supertypes implies the application of the definition of the virtual property given in the supertype referenced by the qualifier. A type whose name is hidden in an enclosing subtype by a redefinition of its type is made visible by qualification with a supertype name. The qualifier consists of only one path item denoting the supertype.

When a virtual type is redefined, the redefined type shall be the same type as the type referenced in the virtuality constraint or a subtype of the referenced type.

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

A property in a subtype shall not be redefined so that it is no longer accessible when the property in the supertype is accessible. In other words, it is not allowed to hide inherited properties in a subtype (e.g., by applying the stereotype <<`private`>> to a property that is <<`public`>> in the supertype).

When a type definition has the stereotype <<`final`>> applied, this type definition shall not be specialized.

Model

When a property does not have a virtuality, this is shorthand for the stereotype <<`final`>> being applied to the property.

When a virtual type does not have a virtuality constraint, the type is used as the virtuality constraint.

The redefinition of a virtual property is allowed in the definition of a subtype of the enclosing type of the virtual property. In the definition of the subtype, the redefined properties of the virtual property include when the virtual property applies to other properties of the subtype inherited from the supertype. A virtual property that is not redefined in a subtype definition has the definition as given in the supertype definition.

Redefinition may be a replacement of the definition (e.g., an operation of the supertype may be replaced with a more specific operation in the subtype) or it may be an extension of the definition (e.g., a port in the subtype may support additional interfaces to those inherited from the supertype). A redefined property may in turn be virtual in the subtype.

When a property replaces the definition inherited from the supertype, the definition of the property in the subtype is used instead of the definition inherited from the supertype.

When a property extends the definition inherited from the supertype, the entities of the property are added to the entities inherited from the supertype. For example, a port on the subtype may specify that it contains additional signals in its port constraints.

How properties are redefined is discussed in the relevant definitions of each property. Table I.4.1 summarizes the impact of redefinition on properties of a type. Column "Constraint" describes constraints on the redefinition in terms of properties of the redefined virtual property. For example, the redefinition of a state shall have at least the connectors of the virtual state it redefines.

Table I.4.1 – Redefinition of virtual properties

Property	Redefinition	Constraint
agent, class, interface, syntype	replaces	shall at least be the virtuality constraint
signal, timer	replaces	shall at least be the type or add parameters
service	replaces	
state	replaces	shall have at least the connectors of the supertype
entry action, exit action, start transition, save, labeled transition, input, spontaneous transition, continuous transition, connect transition	replaces	
channel	replaces	
port, signallist, literal	extends	
Operation	replaces	shall have the same parameter types and may covariantly vary the result type
attribute, agentset	replaces	shall have the same type

I.4.4 Templates

A template is a type definition or operation definition with context parameters. It creates an abstract definition of a type or an operation, leaving properties unspecified. The unspecified properties are denoted by the context parameters, which must all be bound to model elements in order to create a concrete type that can be instantiated or a concrete operation that can be called.

I.4.4.1 Templated entities

A template instantiation is used to define the properties of an entity in terms of a template. A template instantiation denotes a new entity formed by binding actual context parameters to a base identifier. Templated entities may be type definitions, in which case they define a parameterized type, or operation definitions (for operations defined outside of a class or agent), in which case they define a parameterized operation.

Concrete grammar

<*template instantiation*> ::=
    <*base identifier*> <*actual context parameters*>

If <*actual context parameters*> are not supplied for all <*context parameter*> items in the templated entity referenced by the <*base identifier*> and the corresponding <*context parameter*> does not have <*default*> or <*default type*>, as appropriate, the entity is still templated.

<*base identifier*> ::=
    <*identifier*>

The <*identifier*> in a <*base identifier*> shall reference a <*name*> of an entity definition with a <*template*>.

NOTE 1 – Templates and context parameters are defined in clause I.4.4.2.

A <*context parameter*> shall not be used as <*base identifier*> in a <*template instantiation*>.

<*actual context parameters*> ::=
  <{ <*actual context parameter*>+[| **,**]
  | <*named actual context parameter*>+[| **,**] }**>**

<*actual context parameter*> ::=
  [<*template kind*>] {<*primary*> | <*type*> | <*omitted parameter*>}

<*named actual context parameter*> ::=
  [<*template kind*>] <*name*> **=** {<*primary*> | <*type*>}

A <*template instantiation*> with unbound <*actual context parameters*> shall not be used as an <*actual context parameter*>.

An <*omitted parameter*> shall be used as <*actual context parameter*> only if <*default*> or <*default type*> is present in the corresponding <*context parameter*>.

<*template kind*> ::=
  type | **signal** | **timer** | **state** | **operation** | **port**

NOTE 2 – <template kind> may be inserted for additional clarity.

If <*template kind*> is present, the <*actual context parameter*> shall be bound to an entity that is of an appropriate kind, i.e., a type, signal, timer, state or operation.

Model

A <*template instantiation*> is transformed into an anonymous type defined by applying the actual context parameters to the context parameters of the template instantiation denoted by the <*base identifier*>. The definition of this type with anonymous <*name*> is formed by:

1) Copying the definition of the <*base identifier*> into the context where the construct using the <*template instantiation*> occurs and changing the <*name*> to a unique anonymous <*name*>; then in this copy
2) Replacing each occurrence of each <*template name*> by the corresponding <*actual context parameter*>;
3) Removing the <*context parameter*> from the <*context parameter*> list and removing the <*context parameters*> if the <*context parameter*> list is empty;
4) Replacing the <*template instantiation*> by the anonymous <*name*>.
5) If a <*template instantiation*> textually identical to the current <*template instantiation*> has already been transformed, the anonymous <*name*> for that <*template instantiation*> is used rather than creating a new anonymous type.

NOTE 3 – Two textually identical <*template instantiation*> items denote the same type.

In addition to fulfilling any static conditions on the definition denoted by the <*base identifier*>, the anonymous type derived from the <*template instantiation*> shall also fulfil any static condition on the resultant type.

NOTE 4 – For example, the static properties on the usage of a <*template instantiation*> are possibly violated in the following cases:

– Signal context parameters or timer context parameters could introduce non-disjoint triggers, depending on the actual context parameters.
– When an output in a scope unit refers to a port or a channel which is not defined in the nearest enclosing type having ports, instantiation of that type results in an erroneous specification if there is no communication path to that port.
– When a scope unit has an agent context parameter that is used in an output, the existence of a possible communication path depends on which actual context parameter will be used.

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

If <*named actual context parameter*> items are present in the <*actual context parameters*>, then the <*actual context parameter*> items are reordered to match against the <*context parameters*> based on the <*name*> which shall be identical to the <*template name*> of the corresponding <*context parameter*>. Then each <*named actual context parameter*> is replaced by its <*primary*> or <*type*> item, whichever is present. If <*named actual context parameter*> items are present in the <*actual context parameters*>, and an actual context parameter corresponding to a <*context parameter*> is omitted, an <*omitted parameter*> is inserted in the corresponding position in the <*actual context parameters*>.

If an <*actual context parameter*> is an <*omitted parameter*>, the data item provided in the corresponding <*default*> or <*default type*> is used as the <*actual context parameter*>.

The <*base identifier*> of a <*template instantiation*> may also denote an <*operation definition*>. A concrete <*operation definition*> is derived from the parameterized <*operation definition*> in the same manner as for parameterized types.

Mapping

The <*actual context parameters*> represent actualContextParameterList.

An <*actual context parameter*> represents an ActualContextParameter. If the <*actual context parameter*> corresponds to a <*context parameter*> that is not a <*value context parameter*>, then the <*term*> or <*type*> item represents a contextParameter. If the <*actual context parameter*> corresponds to a <*context parameter*> that is a <*value context parameter*>, then the <*term*> or <*type*> item represents a synonymContextParameter.

I.4.4.2 Context parameters

In order for a definition to be used in different contexts, both within the same system specification and within different system specifications, definitions can be parameterized with context parameters. Context parameters are replaced by actual context parameters as defined in clause I.4.4.1.

The following definitions optionally have context parameters: <*class definition*>, <*agent definition*>, <*interface definition*>, <*signallist definition*>, <*timer definition*>, <*operation definition*>.

Context parameters optionally have constraints (which denote required properties any entity referenced by the corresponding actual parameters shall have).

Concrete grammar

<*template*> ::=
    template <*context parameters*>

The scope unit of a definition with a <*template*> defines the <*template name*> for each <*context parameter*> used in the definition. The <*template name*> items of <*context parameters*> are therefore visible in the definition.

<*context parameters*> ::=
    < [<*context parameter*>+[| **,**]] **>**

<*context parameter*> ::=
<*type context parameter*>
| <*input context parameter*>
| <*operation context parameter*>
| <*variable context parameter*>
| <*state context parameter*>
| <*port context parameter*>
| <*value context parameter*>

A <*context parameter*> shall be bound only to an <*actual context parameter*> of the same entity kind that meets the constraint of the <*context parameter*>.

A <*context parameter*> using other <*context parameter*> items in its <*context constraint*> shall not be bound before the other parameters are bound, and if this means there is no possible order for binding the context parameters the specification is not valid.

<*context constraint*> ::=
<*atleast constraint*>
| <*signature constraint*>

Constraints on <*context parameter*> items are specified by the <*context constraint*>.

<*signature constraint*> ::=
: **{** <*entity*>* **}**

A <*signature constraint*> specifies sufficient properties of the definition of the actual context parameter substituted for the context parameter: the definition referenced by the actual context parameter shall be compatible with the signature constraint. A definition is compatible with the signature constraint if it contains entities as specified by the entities in the signature constraint.

An <*entity*> item in a <*signature constraint*> shall be one of the following <*entity*> items: <*class definition*>, <*agent definition*>, <*literal definition*>, <*interface definition*>, <*signallist definition*>, <*port definition*>, <*operation definition*>, <*signal definition*>, <*timer definition*> or <*syntype definition*>. Constructors are identified by an <*operation signature*> (an unnamed constructor is referenced by the <*type identifier*>). A specific <*signature constraint*> shall only contain those entities that are permitted for the kind of definition being constrained.

An <*entity*> in a <*signature constraint*> shall not contain a <*template*>.

<*atleast constraint*> ::=
: <*type*>

An <*atleast constraint*> specifies that the context parameter shall be replaced by an actual context parameter, which is the same type or a subtype of the type referenced in the <*atleast constraint*>.

A <*context parameter*> shall not be used in an <*atleast constraint*>.

<*template name*> ::=
<*name*>

<*type context parameter*> ::=
<*type context parameter kind*> <*template name*> <*stereotype*>* [<*context constraint*>]
[<*default type*>]

If the <*context constraint*> is omitted, any type is allowed as the actual type context parameter.

<*default type*> ::=
= <*type*>

<*type context parameter kind*> ::=
class | **agent** | **interface** | **signallist** | **type**

<*operation context parameter*> ::=
<*template name*> <*operation signature*> <*stereotype*>*

NOTE 1 – An <*operation context parameter*> binds also to constructors. For an unnamed constructor, the name of the type is used in the <*template name*>.

The <*name*> list and the <*default*> in <*parameter*> shall be omitted in the <*operation signature*> of an <*operation context parameter*>.

An <*operation definition*> is compatible with an operation signature if it has the same <*name*>, each <*in parameter*> of the <*operation definition*> is type compatible with, and has the same <*modifier*> as, the corresponding <*in parameter*> of the operation signature, each <*out parameter*> and <*inout parameter*> of the <*operation definition*> has the same <*type*> and the same <*modifier*> as the corresponding <*out parameter*> or <*inout parameter*>, respectively, of the operation signature, and the <*result*> <*type*> is a compatible type.

<*input context parameter*> ::=
<*input context parameter kind*> <*template name*> <*stereotype*>* [<*atleast constraint*>]

<*input context parameter kind*> ::=
signal | **timer**

<*variable context parameter*> ::=
<*template name*> <*context constraint*> <*stereotype*>* [<*default value*>]

<*value context parameter*> ::=
const <*template name*> <*context constraint*> <*stereotype*>* [<*default value*>]

An <*actual context parameter*> substituted for a <*value context parameter*> shall be a <*literal expression*> or a <*variable access*> that is an <*identifier*>.

<*state context parameter*> ::=
state <*template name*> <*state signature*> <*stereotype*>*

An <*actual context parameter*> substituted for a <*state context parameter*> shall identify a <*composite state definition*>. The <*outbound connection points*> of the <*composite state definition*> referenced by the <*actual context parameter*> shall contain all <*name*> items contained in the <*outbound connection points*> of the <*state context parameter*>, if any. The <*inbound connection points*> of the <*composite state definition*> referenced by the <*actual context parameter*> shall contain all <*name*> items contained in the <*inbound connection points*> of the <*state context parameter*>, if any.

NOTE 2 – As the <*name*> items in <*state connection points*> are separated by comma symbols which also separate <*context parameter*> items, the inbound and outbound connection points of a <*state context parameter*> need to be surrounded by parentheses if the <*state context parameter*> is not the last context parameter in the <*context parameters*> or is not followed by a <*stereotype*>.

<*port context parameter*> ::=
port <*template name*> [<*port constraints*>] <*stereotype*>*

An <*actual context parameter*> substituted for a <*port context parameter*> shall reference a <*port definition*>. The <*outbound port constraint*> of the <*port definition*> referenced by the <*actual context parameter*> shall contain all <*identifier*> items contained in the <*outbound port constraint*> of the <*port context parameter*>, if any. The <*inbound port constraint*> of the <*port definition*> referenced by the <*actual context parameter*> shall contain all <*identifier*> items contained in the <*inbound port constraint*> of the <*port context parameter*>, if any.

NOTE 3 – As the <*identifier*> items in <*port constraints*> are separated by comma symbols which also separate <*context parameter*> items, the inbound and outbound port constraints of a <*port context parameter*> need to be surrounded by parentheses if the <*port context parameter*> is not the last context parameter in the <*context parameters*>.

<*default value*> ::=
    `=` <*term*>

The <*default value*> shall be a constant.

Model

The context parameters of a definition that is neither a subtype definition nor defined by binding context parameters in a <*template instantiation*> are the parameters specified in the <*context parameters*>.

The <*context parameter*> items are bound to <*actual context parameter*> items in a <*template instantiation*>. In this binding, occurrences of context parameters inside the templated definition are replaced by the actual context parameters. When binding template names contained in <*context parameter*> items to definitions (that is, deriving their qualifier, see clause I.2.4), local definitions other than the <*context parameters*> items are ignored.

If a scope unit contains <*specialization*>, any omitted actual context parameter in the <*specialization*> is replaced by the corresponding <*context parameter*> of the <*base identifier*> in the <*template instantiation*> and this <*context parameter*> becomes a context parameter of the scope unit.

If the actual parameter substituted for a <*value context parameter*> is an <*expression*> that does not reference a <*variable definition*> with <*immutability*>, there is an implied <*variable definition*> with <*immutability*> and an anonymous <*name*> and the same <*type*> in the context surrounding the templated definition with the <*expression*> as <*default*>.

Mapping

A type definition with <*template*> represents a Classifier, where the <*context parameters*> represent the formalContextParameterList.

A <*context parameter*> represents a FormalContextParameter.

A <*type context parameter*> represents SortContextParameter if <*type context parameter kind*> is `class`, `interface` or `type`, where each <entity> in <*signature constraint*> (if present) represents a signature (a literalSignature, operatorSignature or methodSignature, depending on its kind).

A <*type context parameter*> represents AgentTypeContextParameter if <*type context parameter kind*> is `agent`, where each <entity> in <*signature constraint*> (if present) represents an agentSignature.

A <*type context parameter*> represents InterfaceContextParameter if <*type context parameter kind*> is `signallist`.

The <*template name*> represents contextParameter and the <*atleast constraint*> (if present) represents atLeastClause.

An <*operation context parameter*> represents ProcedureContextParameter. The <*template name*> represents contextParameter, <*atleast constraint*> (if present) represents atLeastClause, and <*operation signature*> (if present) represents procedureSignature.

An <*input context parameter*> represents SignalContextParameter if <*input context parameter kind*> is `signal` or TimerContextParameter if <*input context parameter kind*> is `timer`. The <*template name*> represents contextParameter, and the <*atleast constraint*> (if present) represents atLeastClause.

A <*variable context parameter*> represents VariableContextParameter. The <*template name*> represents contextParameter;<*atleast constraint*> (if present) represents atLeastClause.

A <*value context parameter*> represents SynonymContextParameter. The <*template name*> represents synonymContextParameter, <*atleast constraint*> (if present) represents atLeastClause and <*signature constraint*> (if present) represents sort.

A <*state context parameter*> represents CompositeStateTypeContextParameter. The <*template name*> represents contextParameter, and <*state signature*> (if present) represents compositeStateTypeSignature.

I.4.5 Stereotype

Stereotypes are a mechanism to provide additional information concerning a definition. For example, a stereotype can be used to limit the visibility of an operation in a class, or to indicate which properties of a type may be redefined by a subtype.

Stereotypes may also be used to provide guidance to tools processing a specification. Such stereotypes are not defined in this appendix.

For generality this appendix includes all places where a stereotype is syntactically valid, regardless of whether that stereotype is given an interpretation in this appendix or not.

Concrete grammar

```
<stereotype> ::=
        << [ <stereotype item>+[| ,] ] >>

<stereotype item> ::=
        <stereotype name> [ ( <named value>* ) ]
       | <stereotype name> = <expression>

<stereotype name> ::=
        <name> | <keyword>

<named value> ::=
        [<name> =] <expression>
```

Table I.4.2 summarizes the stereotypes defined throughout this appendix and their applicability.

Table I.4.2 – Stereotypes

Stereotype	Applies to	See clause
`<<comment>>`	anywhere	I.2.2
`<<public>>, <<protected>>, <<private>>`	all entities	I.2.3
`<<abstract>>`	agent, class, operation, constructor	I.4.2
`<<virtual>>, <<redefined>>, <<final>>`	all properties	I.4.3.2
`<<final>>`	agent, class	I.5.1, I.9.1.1
`<<static>>`	operation	I.9.1.5
`<<extern>>`	variable, operation, class	I.11.2

Any stereotypes not discussed specifically in this appendix are not further defined in this appendix and have tool-specific or application-specific meaning.

NOTE – For example, a compiler might define a stereotype <<`inline`>>, to be applied to variable definitions with the meaning that all occurrences of this variable shall be implemented as directly embedded in the defining context rather than as pointers to an object on the heap. For a top-level variable this would imply that the variable is implemented on the stack. As a further example, the compiler may signal an error if it cannot determine that the actual parameters of an operation call can be interpreted in arbitrary order without affecting the result of the interpretation. In this case, the compiler might define the stereotype <<`pure`>> to be applied to the operation call to indicate to the compiler that the results of the actual parameter expressions are independent of the order of their interpretation.

Mapping

A <*stereotype*> defines properties of the containing model element. The <*stereotype*> items apply as specified in Table I.4.2.

The stereotype <<`abstract`>> represents that the isAbstract property of the model element is true.

The stereotype <<`static`>> represents that the isStatic property of the model element is true.

The stereotype <<`const`>> represents that the isConstant property of the model element is true.

The stereotype <<`virtual`>> represents the redefinedElement association of the model element, linking to the model element that is being redefined.

The stereotype <<`final`>> represents that the isLeaf property of the model element is true.

The stereotypes <<`public`>>, <<`protected`>> and <<`private`>> represent the visibility property of the model element being public, protected or private, respectively.

The stereotype <<`comment`>> represents Comment; the <*expression*> represents body; the model element that this stereotype is associated with represents annotatedElement.

Any stereotypes not discussed specifically are not further defined in this Recommendation and have tool-specific or application-specific meaning.

I.5 Structure

The basic structuring concept for a specification is an agent. Each specification consists of a system agent, which may contain other agents.

I.5.1 Agent

An agent represents an independent entity in a system, encapsulating its own thread of control and its own state.

Agents communicate with their environment (the system environment or the external world, or other agents) by means of signal exchanges that avoid simultaneous access to shared data structures.

Agents may contain other agents. These contained agents are placed in agentsets.

All agents have a state machine which, in collaboration with its component agents, defines its behaviour. Signals sent to the agent may be handled by the state machine of the agent, or they may be routed to the components of a composite agent.

Agents may invoke operations in response to signals received. The specific response to signals depends on the state of the agent. The state of the agent is the state of the state machine of the agent and the states of its component agents.

Concrete grammar

<*agent definition*> ::=
    [<*template*>] **`agent`** <*name*> [<*specialization*>] <*stereotype*>* <*entities*>

The <*specialization*> shall contain at most one <*type*>, which shall reference an <*agent definition*>.

An <*entity*> in the <*entities*> of an <*agent definition*> shall be one of the following: <*agent definition*>, <*class definition*>, <*interface definition*>, <*syntype definition*>, <*signal definition*>, <*signallist definition*>, <*timer definition*>, <*variable definition*>, <*agentset definition*>, <*operation definition*>, <*service definition*>, <*channel definition*>, <*constructor definition*>, <*port definition*>.

An <*entity*> of the <*entities*> of an <*agent definition*> shall not have public visibility, except for a <*constructor definition*> or <*port definition*>.

An agent that is not abstract shall contain an unlabelled start transition in its services.

An <*agent definition*> may be contained within another <*agent definition*>. A contained <*agent definition*> shall not be instantiated outside the agent, that is, instances of the contained agent shall be within instances of the enclosing agent. The contained agent may access visible entities of the enclosing agent.

An <*agent definition*> shall not contain, directly or indirectly, a <*variable access*> that references a <*variable definition*> that is contained in an <*agent definition*> that contains this <*agent definition*>, unless this <*variable definition*> has <*immutability*> **`const`** defined. An <*agent definition*> shall not contain, directly or indirectly, an <*assignment*> where the <*location*> references a <*variable definition*> that is contained in an <*agent definition*> that contains this <*agent definition*>. An <*agent definition*> shall not contain, directly or indirectly, an <*agent access*> where the <*agent location*> references an <*agentset definition*> that is contained in an <*agent definition*> that contains this <*agent definition*>.

NOTE 1 – A variable specified by a <*variable definition*> contained in an <*agent definition*> shall be accessed only from the state machine of the agent that contains the <*variable definition*>, unless this variable has <*immutability*> **`const`** defined.

NOTE 2 – Consequentially, an <*agent definition*> shall not contain a <*procedure call*> where the referenced <*operation definition*> accesses variables of the <*agent definition*> that do not have <*immutability*> **`const`**.

A type definition contained within an <*agent definition*> shall not have the visibility <<`public`>> applied.

Any <*attribute definition*>, <*variable definition*>, <*parameter*> or <*result*> with a type defined by an <*agent definition*> shall have **`part`** <*aggregation*>.

If a <*service definition*> of an <*agent definition*> contains a <*constructor definition*> with <*parameters*>, then a <*constructor definition*> of the agent shall contain an <*operation call*> to the constructor of that service in its <*constructor initializer*>.

Model

If no <*specialization*> is given, an <*agent definition*> implicitly specializes the predefined type **`Agent`**.

An agent definition with virtuality is a virtual agent. A virtual agent can be redefined to a subtype in specializations of the enclosing agent definition, subject to virtuality constraints as further discussed in clause I.4.3.2. A redefined agent replaces the virtual agent it redefines.

If no <*constructor definition*> is explicitly present, this is shorthand notation for an unnamed <*constructor definition*> with empty <*parameters*>, empty <*constructor initializer*> and absent <*operation statements*>.

If a service defined in the <*agent definition*> contains a <*default constructor definition*> or an unnamed <*constructor definition*> without <*parameters*> and the constructor defined by this <*constructor definition*> is not called in a <*constructor definition*> of the agent, then a call to this constructor is inserted as the final statement in the <*constructor initializer*> of that <*constructor definition*> of the agent.

Mapping

An <*agent definition*> represents an <<`ActiveClass`>>Class with the isActive and the isConcurrent properties being true. The <*name*> represents name, <*specialization*> (if present) represents general. For each <*entity*> in <*entities*>, an <*attribute definition*> or <*agentset definition*> represents an ownedAttribute; an <*operation definition*> or <*constructor definition*> represents an ownedBehavior; a <*class definition*>, <*interface definition*>, <*signal definition*>, <*timer definition*> or <*syntype definition*> represents a nestedClassifier; a <*channel definition*> represents ownedConnector; a <*port definition*> represents ownedPort.

NOTE 3 – An <<`ActiveClass`>>Class with isActive being true is mapped to an SDL *agent type definition*. An SDL *agent type definition* defines both the SDL agent type, which specifies the properties of agent instances of this agent type, and an SDL interface, which is the type of the identities of the agent instances of this agent type. In this appendix, the phrase "the type defined by an agent definition" refers to the type of the identities of the agent instances of this agent, corresponding to the SDL interface defined in this mapping.

If the <*agent definition*> redefines an <*agent definition*> in a supertype, the redefined <*agent definition*> represents redefinedClassifier.

If the <*entities*> contain a <*service definition*>, a StateMachine is constructed. Each <*service definition*> represents a region. Any <*class definition*>, <*interface definition*>, <*syntype definition*>, <*variable definition*>, <*signal definition*>, <*timer definition*> or <*operation definition*> in that <*service definition*> represent properties of the StateMachine. This state machine represents ownedBehavior.

NOTE 4 – The interpretation of an agent definition with a <*template*> is given in clause I.4.4.2.

I.5.2 System

A specification describes the instantiation of a particular agent. This agent instance is referred to as the *system*, and may contain further agents.

Concrete grammar

<*system*> ::=
    <*singleton agentset*>

I.5.3 Agentset

An agentset specifies a set of agent instances of a particular agent type. The agent instances are interpreted concurrently with each other, the state machine of the agent and other agents in the system.

Concrete grammar

<*agentset definition*> ::=
    <*dynamic agentset*> | <*singleton agentset*>

<*dynamic agentset*> ::=
    `agentset` <*name*> **`:`** <*type*> [<*index type*>] <*stereotype*>* **`;`**

<*index type*> ::=
    `/` [<*type identifier*>] [<*constraint*>]

The <*type*> of a <*dynamic agentset*> shall reference an <*agent definition*>.

The <*constraint*> shall be a <*size constraint*>.

For a <*dynamic agentset*> with a lower bound on its instances established by the <*constraint*> or the type referenced by <*type identifier*>, indicating that initial agent instances will exist in this agentset, the enclosing <*agent definition*> shall contain <*create request*> items in its <*constructor definition*> or in <*constructor definition*> items of its services to create these initial instances.

NOTE 1 – The <*constraint*> establishes a constraint on the number of agent instances that may be present in the agentset and is independent of the manner that an agent instance is accessed in the agentset. The manner the agent instance is accessed is determined by the <*index type*>.

The <*constraint*> in a virtual agentset that is a <*dynamic agentset*> may have a larger upper bound than the <*constraint*> it redefines. A <*dynamic agentset*> shall not be redefined into a <*singleton agentset*>.

The redefinition of a <*dynamic agentset*> shall not change the <*type*> of the virtual agentset it redefines.

<singleton agentset> ::=
    `agentset` <*name*> **`:`** <*type*> <*stereotype*>* <*default*> **`;`**

The <*type*> of a <*singleton agentset*> shall reference an <*agent definition*>.

The <*default*> shall contain an <*operation call*> creating the agent instance in the <*singleton agentset*> and returning this agent instance as its <*result*>.

The redefinition of a <*singleton agentset*> shall not change the <*type*> of the virtual agentset it redefines.

Model

If <*index type*> is not given, or a <*type identifier*> is not given in <*index type*>, then the predefined syntype `Natural` shall be used as <*type identifier*> for the <*index type*> and an <*index type*> is created, if absent.

An <*agentset definition*> with virtuality is a virtual agentset and may be redefined. The <*agentset definition*> in the redefinition replaces the <*agentset definition*> in the supertype of the containing <*agent definition*>.

A <*singleton agentset*> is shorthand for a <*create request*> containing the <*operation call*> of <*default*> as its <*operation call*> and the <*name*> of the <*singleton agentset*> in its <*agent location*> inserted at the end of the constructor for the containing <*agent definition*>.

For a <*dynamic agentset*>, a set of anonymous <*variable definition*> items are created, each with a <*type*> that is the <*type*> of the <*dynamic agentset*> and the <*default*> `null`, up to the number of elements as given by the <*constraint*> in the <*index type*>. Each variable corresponding to an anonymous <*variable definition*> item is associated with an element of the type referenced by the <*index type*>.

NOTE 2 – When agent instances of the agent identified by the type of the agentset are created, their identities are associated with the variables implied by the agentset. The phrase "an agent instance in the agentset" refers to an agent instance created as specified by the agentset such that its identity is associated with a variable implied by the agentset. Each such variable implied by the agentset corresponds to an element of the index type and can be identified by that element.

For a <*singleton agentset*> an anonymous <*variable definition*> is created with a <*type*> which is the type of the <*singleton agentset*> and the <*default*> `null`.

Mapping

An <*agentset definition*> represents Property with <*name*> representing name, <*type*> representing type, and aggregation being composite. For a <*singleton agentset*>, initialNumber is 0, lowerValue is 1 and upperValue is 1. For a <*dynamic agentset*>, initialNumber is 0, and if there is a <*constraint*> on <*index type*>, the lowerValue and upperValue correspond to the lower and upper bounds of the <*constraint*>; otherwise, if <*constraint*> is not given, lowerValue and upperValue correspond to the lower and upper bounds on <*type*>, if any. The <*expression*> in <*default*>, if present, represents defaultValue.

I.6 Communication

Communication is a fundamental concept in a specification and is allowed between agent instances provided communication paths (channels) exist between the agent instances. Channels may be established between the ports on an agent. Any two ports may be connected if they are directly visible to each other. Ports and channels may impose constraints on the items that may be communicated across them. Communication between agent instances is asynchronous and discreet, with signals being the individual items of communication.

I.6.1 Port

Ports are defined in agent definitions and represent connection points for channels connecting the state machine of the agent or contained agents with its environment, and for connecting the state machine of the agent with its contained agents.

Ports allow constraints to be specified on the communication that is permitted to flow in and out of the agent. This also constrains the types of agents that may be connected (via channels) to a port.

Concrete grammar

<*port definition*> ::=
    `port` <*name*> [<*port constraints*>] <*stereotype*>* ;
  | **`port`** <*port constraints*> <*stereotype*>* ;

A <*port definition*> that does not have a <*name*> shall have either an <*inbound port constraint*> or an <*outbound port constraint*> and it shall have only one <*identifier*> in the <*identifier*> list of its <*inbound port constraint*> or <*outbound port constraint*>, respectively.

<*port constraints*> ::=
    <*inbound port constraint*> [<*outbound port constraint*>]
  | <*outbound port constraint*> [<*inbound port constraint*>]

<*inbound port constraint*> ::=
    `in` { [<*identifier*>+[| **`,`**]] | **`(`** [<*identifier*>+[| **`,`**]] **`)`** }

<*outbound port constraint*> ::=
    `out` { [<*identifier*>+[| **`,`**]] | **`(`** [<*identifier*>+[| **`,`**]] **`)`** }

If the same <*identifier*> is used in several <*signallist definition*> items referenced in <*port constraints*>, these reference the same signal.

An <*identifier*> in the <*identifier*> list of an <*inbound port constraint*> or <*outbound port constraint*> shall reference only <*signallist definition*> or <*signal definition*> items.

Model

A <*port definition*> with virtuality is a virtual port. A virtual port may be redefined in a specialization. A redefined port is extended in the subtype. The <*type identifier*> list in an inbound or outbound port constraint of the redefined port is the union of the <*type identifier*> lists of the inherited inbound or outbound port constraints, respectively, and the inbound or outbound port constraints, respectively, in the redefinition.

If an <*identifier*> in an <*inbound port constraint*> or <*outbound port constraint*> references a <*signallist definition*> this is shorthand for the <*identifier*> items in the referenced <*signallist definition*>.

A <*port definition*> without a <*name*> is shorthand for a <*port definition*> having a <*name*> that is the <*name*> of the <*identifier*> in its <*inbound port constraint*> or <*outbound port constraint*>.

Mapping

A <*port definition*> represents a Port with isBehavior being false. If present, the <*name*> represents name; otherwise, an anonymous name is created. Each <*type identifier*> in its <*inbound port constraint*> represents a name in the providedInterface. Each <*type identifier*> in its <*outbound port constraint*> represents a name in the requiredInterface.

I.6.2 Channel

A channel is a directed communication path between two ports. Channels convey the signals used for communication from one port to another. The signals conveyed in one direction on a channel cannot be conveyed in the opposite direction unless explicitly stated.

Concrete grammar

<*channel definition*> ::=
  `channel` [<*name*>] <*stereotype*>* <*channel end*> <*channel end*> **`;`**

<*channel end*> ::=
  <*source channel end*>
  | <*target channel end*>

At least one of the <*channel end*> items shall be a <*target channel end*>.

A channel is said to be connected to the entity referenced in a <*channel destination*> in its <*source channel end*> or <*target channel end*>.

NOTE 1 – It is permissible that several channels exist between the same two channel ends.

A <*port definition*> with an <*inbound port constraint*> and an <*outbound port constraint*> may appear as both <*source channel end*> and <*target channel end*> of the same <*channel definition*>.

If the <*source channel end*> and the <*target channel end*> reference <*port definition*> items of the same <*agent definition*>, the <*channel definition*> shall be unidirectional.

The entities referenced by <*channel end*> items shall be defined in the same scope unit in which the <*channel definition*> is contained.

<*source channel end*> ::=
  `from` <*channel destination*>++[| **`,`**]

<*target channel end*> ::=
  `to` <*channel destination*>+++[| **`,`**] [<*channel constraint*>]

<*channel destination*> ::=
  <*agent port*> | <agentset *port*> | <*service destination*>

<*channel constraint*> ::=
  `with` <*identifier*>+[| **`,`**]

The <*identifier*> items in <*channel constraint*> shall reference either <*signallist definition*> or <*signal definition*> items.

NOTE 2 – It is permissible that the same signal <*identifier*> occurs in several <*channel constraint*> items.

<*agent port*> ::=
  <*identifier*>

The <*identifier*> in an <*agent port*> shall reference a <*port definition*> in the containing <*agent definition*>.

<agentset *port*> ::=
  <*identifier*> **`.`** <*identifier*>

The first <*identifier*> in <agentset *port*> shall reference an <*agentset definition*>; the second <*identifier*> shall reference a <*port definition*> in the <*agent definition*> that is the <*type*> of the <*agentset definition*>.

If a <*port definition*> is referenced in a <*channel end*> of the <*channel definition*>, the <*channel definition*> shall be compatible with the relevant <*inbound port constraint*> or <*outbound port constraint*> of the <*port definition*> (see below).

<*service destination*> ::=
  <*identifier*> | <*any service*>

<*any service*> ::=
  `*`

The <*identifier*> in a <*channel destination*> that is a <*service destination*> shall reference a <*service definition*> of the containing <*agent definition*>. If a <*target channel end*> contains a <*service destination*> that is an <*identifier*>, the signals referenced by <*type identifier*> items in <*channel constraint*> shall not be used in any <*service definition*> not referenced in the <*channel destination*>, even if these <*type identifier*> items may be referenced in the <*channel constraint*> of another <*channel definition*> connected to that other service.

Channels between ports on an agent (an <*agent port*>) and an agentset (an <agentset *port*>) shall be explicitly specified. Channels from a port to a service of the agent (a <*service destination*>) shall be explicitly specified. In particular, a channel to a service carrying a signal shall be specified, if the service can consume that signal and a channel exists to another service for the same signal. It is permitted to specify channels carrying a signal to services of the agent even if the agent does not consume that signal.

A <*channel definition*> with the <*identifier*> of a <*port definition*> in its <*channel destination*> where the other <*channel destination*> is not a <*service destination*> shall be compatible with the <*port constraints*> of the <*port definition*>.

A <*channel definition*> is compatible with the <*port constraints*> if:

a) the port constraint is an inbound port constraint and
   1. if the channel has a channel constraint in its target channel end and the port is referenced in the target channel end, the port constraint conveys the signals in the channel constraint, or
   2. the other channel destination of the channel is a source channel end and references the port on an agent of the type referenced by a <*type identifier*> in the inbound port constraint or a supertype of this agent, or references a port such that all signals conveyed by that port are conveyed in the inbound port constraint;

b) the port constraint is an outbound port constraint and
   1. if the channel has a channel constraint in its target channel end and the port is referenced in the other channel end, the outbound port constraint conveys the signals in the channel constraint, or
   2. if the channel has a channel constraint in its target channel end and the other channel destination of the channel is a target channel end and references the port on an agent of the type referenced by a type identifier in the outbound port constraint or a subtype of this agent, or references a port which conveys all signals conveyed in the outbound port constraint.

Model

If a <*channel destination*> of a <*channel end*> is <*any service*>, this is shorthand for a set of <*channel destination*> items with <*service destination*> for each of the services of the containing agent definition, with the <*identifier*> of the service as <*identifier*>. These <*channel destination*> items are inserted into the <*channel end*> instead of the <*any service*>.

If the <*name*> is omitted from a <*channel definition*>, the channel is implicitly and uniquely named.

A <*channel definition*> where both <*channel end*> items are a <*target channel end*> is split into two separate anonymous <*channel definition*> items, where one <*channel end*> is a <*target channel end*> and the other <*channel end*> is a <*source channel end*> for the same <*destination*> lists and <*channel constraint*> items.

A <*channel definition*> having an <agentset *port*> as the <*channel destination*> of both <*channel end*> items is shorthand for individual channels from each of the agent instances in the referenced agentset to all other agent instances in the set, including the originating agent. Any resulting bidirectional <*channel definition*> items connecting an <*agent definition*> in the agentset to that <*agent definition*> itself is split into two unidirectional <*channel definition*> items as defined above.

If a <*channel constraint*> is omitted from a <*target channel end*>, this <*channel constraint*> is derived as the union of the signals contained in signallist items referenced in the <*outbound port constraint*> items of all ports referenced in <*channel destination*> items of the <*source channel end*> and the signals used in any service referenced in <*channel destination*> items of the <*source channel end*>.

If an <*identifier*> in a <*channel constraint*> references a <*signallist definition*>, this is shorthand for the <*identifier*> items in the referenced <*signallist definition*>. If the same signal <*identifier*> is contained in multiple referenced <*signallist definition*> items, these identify the same signal.

A <*channel definition*> with virtuality is a virtual channel and may be redefined. The <*channel definition*> in the redefinition replaces the <*channel definition*> in the supertype of the containing <*agent definition*>.

Mapping

A <*channel definition*> represents Connector with the delay property being true. The <*name*> (if present) represents name; otherwise an anonymous name is created.

Each <*channel end*> represents a ConnectorEnd referenced by the end property of the Connector. Each <*type identifier*> in the <*type identifier*> list of <*channel constraint*> represents an InformationItem conveyed by the corresponding InformationFlow in the direction defined by the <*channel end*>.

Each <*channel destination*> represents the role of the respective ConnectorEnd. The <*qualified name*> of an <*agent port*> represents name. The first <*qualified name*> of an <agentset *port*> represents partWithPort; the second <*qualified name*> represents name. The <*qualified name*> of a <*service destination*> represents name.

I.6.3 Signal

A signal definition specifies that an instance of a designated class may be conveyed in a communication.

Concrete grammar

<*signal definition*> ::=
    [<*template*>] `signal` <*stimulus definition item*>+[| `,`] `;`

An <*identifier*> with the <*name*> `stop` shall not be present in a <*stimulus definition item*>.

A <*context parameter*> in the <*template*> of a <*signal definition*> shall be a <*type context parameter*>.

<*stimulus definition item*> ::=
    <*type identifier*> [<*parameters*>] <*stereotype*>*

If a <*stimulus definition item*> contains <*parameters*>, the <*type identifier*> shall not contain a <*qualifier list*>. A <*parameter*> shall be an <*in parameter*> and shall not have `ref` <*aggregation*> nor shall its type contain, directly or indirectly, an <*attribute definition*> with `ref` <*aggregation*>.

If a <*stimulus definition item*> does not contain <*parameters*>, the <*type identifier*> shall reference a <*class definition*>. The referenced <*class definition*> shall not contain, directly or indirectly, an <*attribute definition*> with `ref` <*aggregation*>. The <*type identifier*> shall not reference a primitive type.

NOTE 1 – Consequentially, a signal conveyed to an agent will never contain references to data items owned by another agent.

The <*class definition*> referenced by <*type identifier*> without <*parameters*> in a redefinition shall be a subtype of the class definition referenced in the virtual property it redefines. If the <*stimulus definition item*> contains <*parameters*>, a redefinition shall contain at least the <*parameter*> items contained in the virtual property it redefines.

Model

When more than one <*stimulus definition item*> occurs in a <*signal definition*>, a separate <*signal definition*> is created for each <*stimulus definition item*> and the original <*stimulus definition item*> is deleted from the <*signal definition*>.

For a <*stimulus definition item*> with <*parameters*>, a <*class definition*> with the same <*name*> as the <*name*> in the <*type identifier*> is constructed. For each <*parameter*>, an <*attribute definition*> with the corresponding <*type*> is constructed. The <*name*> of the <*parameter*> is used as the <*name*>; if no <*name*> is present, an anonymous <*name*> is used. A constructor for this class is created with <*parameters*> as signature and an assignment from each <*parameter*> to the corresponding <*name*> referencing a <*variable definition*>, as implicitly derived from the attribute definition (see clause I.9.3.1), in the <*constructor initializer*>.

NOTE 2 – If a <*parameter*> does not contain a <*name*>, an <*input variable*> corresponding to this <*parameter*> is not available.

A <*signal definition*> with virtuality is a virtual signal and may be redefined. The <*signal definition*> in the redefinition replaces the <*signal definition*> in the supertype of the containing <*agent definition*>.

Mapping

Each <*stimulus definition item*> in <*signal definition*> represents a Signal. The <*type identifier*> represents name. There shall be a <*class definition*> with the same <*type identifier*> that is either already existing or is constructed as described in the Model above. Each <*attribute definition*> in this <*class definition*> represents an ownedAttribute.

NOTE 3 – The interpretation of a signal definition with a <*template*> is given in clause I.4.4.2.

I.6.4 Signallist

A signallist groups a set of signals and is used in a channel constraint or port constraint to denote that all signals specified in the signallist definition are included in the channel constraint or port constraint.

The defining context of the signals in a signallist is the scope unit containing the signallist. The signals in a signallist are visible where the signallist is visible.

Concrete grammar

<*signallist definition*> ::=
    [<*template*>] **`signallist`** <*name*> <*stereotype*>* **`=`** <*identifier*>+[| **`,`**] **`;`**

A <*context parameter*> in the <*template*> of a <*signallist definition*> shall be a <*type context parameter*> or <*input context parameter*>.

The <*name*> of the <*signallist definition*> shall not be contained directly or indirectly in the <*identifier*> list of a <*signallist definition*> referenced in the <*identifier*> list.

Model

If an <*identifier*> references a <*signallist definition*>, this is shorthand for including each element of the <*identifier*> list of the referenced <*signallist definition*> instead of the signallist <*identifier*>.

A <*signallist definition*> with virtuality is a virtual signallist. A virtual signallist may be redefined in a specialization. A redefined signallist is extended in the subtype. The <*identifier*> items of the redefined signallist are the union of the inherited <*identifier*> items and the <*identifier*> items in the redefinition.

Mapping

The representation of a <*signallist definition*> is given in the Model above.

NOTE – The interpretation of a signallist definition with a <*template*> is given in clause I.4.4.2.

I.7 State machine

A state machine is the basic behavioural abstraction used for agents. Each agent has an implicit or explicit state machine composed of states and the transitions between the states. The behaviour of the state machine is determined by its current state and input. The behaviours (actions) are specified on the transitions between the states. Either implicitly or explicitly, a transition is defined for every stimulus in the valid input signal set in every state. That is, for every state in an agent, a transition is defined for every stimulus that may be received by that agent.

I.7.1 Service

A service is a partitioning of the state machine of an agent. Each service has a state machine, and the composition of the state machines of these services forms the complete state machine of the agent. The state machine of a service has an interpretation of interleaving transitions with the state machines of other services of the same containing agent. At any given time, each service is in one of the states of that service, or (for one of the services only) in a transition, or has completed and is waiting for other services to complete. Each transition runs to completion.

Concrete grammar

<*service definition*> ::=
    `service` [<*name*>] <*stereotype*>* { <*entities*> | <*service instantiation*> }

<*service instantiation*> ::=
    `=` { <*identifier*> | <*template instantiation*> }`;`

An <*entity*> in the <*entities*> of a <*service definition*> shall be one of the following: <*class definition*>, <*interface definition*>, <*signallist definition*>, <*syntype definition*>, <*signal definition*>, <*timer definition*>, <*operation definition*>, <*start transition*>, <*state definition*>, <*transition*>, <*labeled transition*>, <*constructor definition*> and <*variable definition*>.

An <*entity*> in a <*service definition*> shall not contain a <*template*>.

A <*constructor definition*> in a <*service definition*> shall not contain <*operation statements*>.

A <*service instantiation*> shall reference a visible <*service definition*> in its <*identifier*> or the <*identifier*> of its <*template instantiation*>.

Model

A <*service definition*> with virtuality is a virtual service and may be redefined. The <*service definition*> in the redefinition replaces the <*service definition*> in the supertype of the containing <*agent definition*>.

When a <*service definition*> contains a <*service instantiation*>, the <*entities*> of the referenced <*service definition*> replace the <*service instantiation*>.

If <*service definition*> contains a <*signal definition*> or <*timer definition*>, these definitions are moved to the containing <*agent definition*> and they are given an anonymous <*name*>.

Mapping

A <*service definition*> represents Region. The name, if present, represents name; otherwise, an anonymous name is created. Each <*state definition*> represents a subvertex; each <*transition*>, <*start transition*>, and each <*labeled transition*> represents a transition; each <*attribute definition*> represents ownedAttribute.

I.7.2 State

A state machine in a state awaits stimuli, which are either signals sent to the agent containing the state machine, signals sent from other services or from contained agents of that agent, timer signals or signals sent from within the state machine. When a state detects an enabled stimulus and the guard condition, if any, is satisfied, and the stimulus arrived via the designated path, if any, the transition defined for this stimulus is interpreted. Alternatively, a stimulus may be saved to be handled in another state, or transitions based on conditions being satisfied are interpreted. After interpretation of a transition, the state machine enters a state, which may be the activating state, or a different state.

Concrete grammar

```
<state definition>  ::=
        <basic state definition>
      | <composite state definition>

<basic state definition>  ::=
        state <name> <stereotype>* ;
```

A virtual basic state definition may be redefined to a composite state definition. A virtual composite state definition shall be redefined only to a composite state definition.

```
<transition>  ::=
        for state { <state list> | ( <state list> ) }
          {{ {    <input>
                 | <save>
                | <continuous transition>
                | <spontaneous transition>
                | <labeled transition>
                | <connect transition> }* } }
        | ;
```

A <*transition*> shall contain at most one <*asterisk input list*> (see clause I.7.3.2). A <*transition*> shall contain at most one <*asterisk save list*> (see clause I.7.3.6). A <*transition*> shall not contain both an <*asterisk input list*> and an <*asterisk save list*>.

```
<state list>  ::=
        <name>+[| ,] | <asterisk state list>

<asterisk state list>  ::=
       * [ ( <name>+[| ,] ) ]
```

The <*name*> items in an <*asterisk state list*> shall be distinct and shall be contained in other <*state list*> items or in <*state definition*> items of the enclosing <*entities*> or of the <*entities*> of a supertype of the enclosing agent.

A <*name*> in a <*state list*> or <*asterisk state list*> represents the behaviour of either a basic state or a composite state. The term "within a composite state" when applied to a state means that this state is part of a service defined within a <*composite state definition*>.

Model

A <*state definition*> with virtuality is a virtual state and may be redefined. The <*state definition*> in the redefinition replaces the <*state definition*> in the supertype of the containing <*agent definition*>.

If a <*service definition*> does not contain a <*state definition*> (and no state is inherited from the supertype of the containing agent), this is shorthand for a <*basic state definition*> with an anonymous <*name*>. The unlabelled <*start transition*> of this <*service definition*> contains a <*named nextstate*> as its only <*statement*> in the <*transition statements*> such that its <*identifier*> references that state. This anonymous state is not inherited in a specialization.

For every state referenced in a <*state list*> or <*state definition*>, a <*transition*> is created with an <*input*> with <*stimulus*> **`stop`** and a <*stop statement*> as <*transition statements*>, unless an <*input*> or <*save*> with **`stop`** as <*stimulus*> is already defined for this state.

When the <*state list*> of a <*transition*> contains more than one <*name*>, a copy of that <*transition*> is created for each such <*name*>. Then the <*transition*> is replaced by these copies.

When several <*transition*> items contain the same <*name*> in their <*state list*>, these <*transition*> items are combined into one <*transition*> having that <*name*>.

A <*transition*> with an <*asterisk state list*> is transformed to a set of <*transition*> items, one for each <*name*> of a state of the enclosing state machine, except for those <*name*> items contained in the <*asterisk state list*>.

Mapping

A <*basic state definition*> represents State. The <*name*> represents name. If the <*basic state definition*> redefines a <*basic state definition*> in a supertype, the redefined <*basic state definition*> represents redefinedClassifier.

The <*input*>, <*save*>, <*continuous transition*>, <*spontaneous transition*>, <*labeled transition*> and <*connect transition*> represent a Transition, as further described below. If <*state list*> contains a name, the Transition is the outgoing property of the Vertex represented by the name. The source property of the Transition is the Vertex identified in the <*state list*>. The target property of the Transition is the Vertex identified by the <*transition terminating statement*> in the <*statements*> representing the effect.

I.7.3 Transition

The transitions of a state machine serve two main purposes: they specify the actions that should happen given a current state and a stimulus, and they specify the next state that should be entered at the completion of the transition.

I.7.3.1 Start transition

Every state machine has a transition which specifies the actions that happen when the interpretation of the state machine begins. A start transition does not define a state and cannot be performed again after a transition.

Concrete grammar

<*start transition*> ::=
    `start` [<*name*>] <*stereotype*>* <*transition statements*>

All potentially instantiated agents shall have a <*start transition*>. There shall be exactly one <*start transition*> without <*name*> in a <*service definition*>.

A <*start transition*> with a <*name*> shall be within the <*entities*> of a <*composite state definition*>. The <*name*> shall reference a label in the <*inbound connection points*>.

Model

A <*start transition*> with virtuality is a virtual start and may be redefined. The <*start transition*> in the redefinition replaces the <*start transition*> in the supertype of the containing <*agent definition*>.

NOTE – A <*start transition*> contained in a <*composite state definition*> is defined in clause I.7.4.1.

Mapping

A <*start transition*> represents a Transition that is the outgoing property of a start Pseudostate with the <*name*> representing name, if present, and the <*transition statements*> representing the effect.

If the <*start transition*> redefines a <*start transition*> in a supertype, the redefined <*start transition*> represents redefinedTransition.

I.7.3.2 Input transition

For each state, an input transition defines what actions take place when a given stimulus occurs (or one of a set of stimuli occurs). Associated with each state, the transition for a stimulus may be guarded by a condition. Also associated with each state, a stimulus may be given a priority. The guards and priorities of each state are independent of each other.

Concrete grammar

<*input*> ::=
    `input` { <*trigger*> | `(` <*trigger*> `)` } [<*guard*>] [<*priority*>] <*stereotype*>*
        <*transition statements*>

Within the <*transition statements*> of the <*input*>, the implicitly declared variable **`input`** is accessible and is associated with the stimulus received (a signal or timer). The variable **`input`** shall not be the <*location*> of an <*assignment*>.

If in the <*transition statements*> **`input`** occurs as the <*target*> of a <*method call*>, an <*operation definition*> referenced by the <*identifier*> of the <*invocation*> shall be present in each type definition referenced by every <*stimulus*> in <*trigger*> and shall have the same <*result*> type.

If in the <*transition statements*> **`input`** is the <*expression*> of an assignment or is an <*actual parameter*>, the type definition items referenced by all <*stimulus*> items occurring in <*trigger*> shall have a common (direct or indirect) supertype and this common supertype shall be the <*type*> of the <*location*> of the <*assignment*> or the <*parameter*>, respectively.

If in the <*transition statements*> **`input`** is the <*expression*> of an <*output clause*>, each <*stimulus*> occurring in <*trigger*> shall be conveyed by an <*outbound port constraint*> of a <*port definition*> contained in the containing <*agent definition*>.

A virtual input shall not contain an <*asterisk input list*>. A virtual input may be redefined to a priority input, to an input or to a save. The virtual input and the redefined input shall have the same <*stimulus*> and if one <*stimulus*> has a <*path*>, the other shall also have a <*path*>.

The <*expression*> of the <*guard*> shall not reference the implicitly declared **`input`** variable or an <*input variable*> used in the <*trigger*>.

<*trigger*> ::=
    <*stimulus list*> | <*asterisk input list*>

<*stimulus list*> ::=
    <*stimulus*>+[| `,`]

The <*type identifier*> of a <*stimulus*> occurring in a <*stimulus list*> shall reference a <*stimulus definition item*> of a <*signal definition*> or a <*timer definition*>.

<*stimulus*> ::=
    <*type identifier*> [`(` <*input variable*>* `,` `)`] [<*path*>]
    | <*state timer*>

The optional <*path*> of a <*stimulus*> shall reference a port of the enclosing agent. The referenced <*port definition*> shall include the signal referenced in the <*stimulus*> in its <*inbound port constraint*>.

If a <*stimulus*> has no associated <*path*>, this <*stimulus*> shall not appear without a <*path*> in a <*trigger*> in another <*input*> or <*save*> for the same state. If a <*stimulus*> has an associated <*path*>, this <*stimulus*> shall not appear with the same <*path*> in a <*trigger*> in another input or save for the same state.

The <*type identifier*> in a <*stimulus*> shall reference a <*name*> of a <*stimulus definition item*> or the <*name*> **`stop`**.

When the <*type identifier*> in a <*stimulus*> references a **`stop`**, the <*transition statements*> of the containing <*input*> shall terminate (directly or indirectly) with a <*stop statement*>.

NOTE 1 – While a **`stop`** signal may be saved, it shall not be saved in every state of the state machine of the agent.

The <*type identifier*> of each <*stimulus*> in <*stimulus list*> shall reference a <*stimulus definition item*> of a <*signal definition*> or a <*timer definition*>.

If the <*type identifier*> in a <*stimulus*> references a <*stimulus definition item*> of a <*timer definition*>, <*path*> shall not be present.

If the <*identifier*> in a <*path*> of a <*stimulus*> references a <*channel definition*>, it shall identify a channel connected to the enclosing state machine.

<*state timer*> ::=
    <*named state timer*> | <*unnamed state timer*>

<*named state timer*> ::=
    <*set*>

<*unnamed state timer*> ::=
    `timer` <*default*>

The <*default*> in a <*timer set clause*> shall be omitted only if the referenced <*timer definition*> has a <*default*>. The <*default*> in an <*unnamed state timer*> shall not be omitted. The <*expression*> in the default shall be an <*operand*>.

<*input variable*> ::=
    <*name*> <*stereotype*>*

The <*name*> of an <*input variable*> shall reference an accessor (see clause I.9.1.6) defined in the type definitions referenced by the <*type identifier*> of the containing <*stimulus*>.

An <*input variable*> shall not be the <*location*> of an <*assignment*> in the <*transition statements*>.

<*asterisk input list*> ::=
    `*`

Model

A <*name*> in a <*path*> of a <*stimulus*> referencing a <*channel definition*> is shorthand for a port or a set of ports. For each port that is a <*source channel end*> of the <*channel definition*>, a copy of the <*input*> that contained the <*stimulus*> is created and the <*stimulus*> with the port referenced in <*path*> is used as <*trigger*>. The <*stimulus*> is removed from the original <*input*>, and if the <*stimulus list*> is empty thereafter, the <*input*> is deleted.

When more than one <*stimulus*> occurs in the <*trigger*> of an <*input*>, a copy of the <*input*> is made for each <*stimulus*>, using that <*stimulus*> as <*trigger*> and inserted into the containing <*transition*>.

When the <*trigger*> is an <*asterisk input list*>, a copy of the <*input*> is made for each <*type identifier*> of the complete valid input signal set of the containing <*service destination*> except for any <*type identifier*> referencing a <*signal definition*> contained in any other <*input*> or <*save*> of the containing <*transition*>, using that <*type identifier*> as <*stimulus*> in <*trigger*>, and inserted into the containing <*transition*>. If the <*trigger*> has a <*priority*>, the created input has the same <*priority*>.

NOTE 2 – If the <*asterisk input list*> is applied to a substate within a composite state, the substate has an input or save for every receivable signal, and therefore in this substate it is not possible to trigger a transition on a composite state that uses the composite state.

A <*stimulus*> where the <*identifier*> in <*type identifier*> references a <*signallist definition*> is shorthand notation for a list of <*stimulus*> items, one for each <*signal definition*> referenced in the <*identifier*> list of the <*signallist definition*>.

An <*input*> with virtuality is a virtual input and may be redefined. The <*input*> in the redefinition replaces the <*input*> in the supertype of the containing <*agent definition*>.

An anonymous <*variable definition*> is created in the containing <*agent definition*> with aggregation **`part`**, and a <*type*> which is the class referenced by the <*stimulus definition item*> referenced by the <*type identifier*> in the <*stimulus*>.

For each <*attribute definition*> in the <*stimulus definition item*> referenced by the <*type identifier*> in the <*stimulus*>, an anonymous <*variable definition*> is created in the <*agent definition*> containing the <*input*>, with the <*type*> and <*modifier*> of the <*attribute definition*>.

If the <*stimulus*> does not contain <*input variable*> items, all references to the **`input`** variable in the <*transition statements*> of <*input*> are replaced with references to the variable corresponding to the stimulus.

If the <*stimulus*> contains <*input variable*> items, each reference to an <*input variable*> in the <*transition statements*> of the <*input*> is replaced with a <variable *access*> to the anonymous variable corresponding to the attribute referenced by the <*input variable*>.

An <*assignment*> is inserted as the first <*statement*> in the <*transition statements*> of <*input*> with a <*constructor call*> to the default constructor or the unnamed nullary constructor of the class corresponding to the <*signal definition*> referenced by the <*type identifier*> in the <*stimulus*> as the <*expression*> and the anonymous variable corresponding to the stimulus as <*location*>. Following this a sequence of <*assignment*> items is inserted in the <*transition statements*>, one for each <*attribute definition*> in the referenced <*stimulus definition item*> with the corresponding attribute of the anonymous variable corresponding to the stimulus as <*location*> and the anonymous variable corresponding to the attribute as <*expression*>. The <*input variable*> list of the <*stimulus*> is replaced by a new <*input variable*> list with an <*input variable*> with the <*name*> of the corresponding attribute of the class referenced by the <*stimulus definition item*> referenced by the <*type identifier*> in the <*stimulus*> for such an attribute.

Mapping

An <*input*> represents a Transition. The <*trigger*> represents trigger; the <*expression*> in <*guard*> (if present) represents guard; the <*expression*> in the <*priority*> (if present) represents priority; and the <*transition statements*> represent the effect. The <*path*> (if present) represents port. The <*input variable*> items after application of the Model in clause I.7.3.2 (which correspond to the attributes in the class definition referenced by the stimulus definition item referenced by the stimulus) represent variableList.

If the <*input*> redefines an <*input*>, <*save*> or priority input in a supertype, the redefined <*input*>, <*save*> or priority input, respectively, represents redefinedTransition.

Each <*stimulus*> in the <*stimulus list*> represents a Trigger. The event of Trigger is a SignalEvent or TimeEvent, depending on whether the <*type identifier*> references a signal or a timer, respectively.

I.7.3.3 Guard

A guard makes it possible to impose an additional condition on the consumption of a stimulus, beyond its reception, or to impose a condition on a spontaneous transition.

Concrete grammar

<*guard*> ::=
    `if` <*expression*>

Mapping

NOTE – The mapping for a <*guard*> is given in clauses I.7.3.2, I.7.3.5 and I.7.3.7.

I.7.3.4 Priority

Concrete grammar

<*priority*> ::=
    `priority` <*expression*>

A priority input occurs when the containing <*input*> has a <*priority*>.

The <*expression*> in <*priority*> shall be a constant with the predefined syntype `Natural` as type.

A virtual priority input may be redefined to a priority input, to an input or to a save. The virtual priority input and the redefined priority input shall have the same <*stimulus*> and if one <*stimulus*> has a <*path*>, the other shall also have a <*path*>.

Mapping

NOTE – The mapping for <*priority*> is given in clauses I.7.3.2, I.7.3.5 and I.7.3.7.

If the priority input redefines an <*input*>, <*save*> or priority input in a supertype, the redefined <*input*>, <*save*> or priority input, respectively, represents redefinedTransition.

I.7.3.5 Continuous transition

A continuous transition allows interpretation of a transition when a certain condition is fulfilled. A continuous transition interprets a Boolean expression and the associated transition is interpreted when the expression returns the predefined `Boolean` value `true`.

Concrete grammar

<*continuous transition*> ::=
    <*guard*> [<*priority*>] <*stereotype*>* <*transition statements*>

If several virtual continuous transitions exist in a state, then each of these shall have a distinct priority. If only one virtual continuous transition exists in a state, it is allowed to omit the priority.

Model

A <*continuous transition*> with virtuality is a virtual continuous transition and may be redefined. The <*continuous transition*> in the redefinition replaces the <*continuous transition*> in the supertype of the containing <*agent definition*>.

Mapping

A <*continuous transition*> represents a Transition. The trigger is a Trigger; the event of the trigger is a ChangeEvent. The <*expression*> in <*guard*> represents guard; the <*expression*> in the <*priority*> (if present) represents priority; the <*transition statements*> represent the effect.

If the <*continuous transition*> redefines a <*continuous transition*> in a supertype, the redefined <*continuous transition*> represents redefinedTransition.

I.7.3.6 Save

A save specifies a set of stimuli whose instances are not relevant to the agent in the state with the save, but which need to be dealt with in future processing.

Concrete grammar

<*save*> ::=
    save { <*saved trigger*> | **(** <*saved trigger*> **)** } <*stereotype*>* **;**

A virtual save shall not contain an <*asterisk save list*>. A virtual save may be redefined to a priority input, to an input or to a save. The virtual save and the redefined save shall have the same <*stimulus*> and if one <*stimulus*> has a <*path*>, the other shall also have a <*path*>.

<*saved trigger*> ::=
    <*stimulus list*> | <*asterisk save list*>

<*asterisk save list*> ::=
    *****

A <*stimulus*> in the <*stimulus list*> shall not contain <*input variable*> items.

The optional <*path*> in a <*stimulus*> shall reference a port of the enclosing agent. The referenced <*port definition*> shall include the signal referenced in the <*stimulus*> in its <*inbound port constraint*>.

If a <*stimulus*> in a <*save*> does not have a <*path*>, a <*stimulus*> in the <*saved trigger*> shall not appear without a <*path*> in an <*input*> for the same state. If an <*input*> has a <*path*>, a <*stimulus*> in the <*saved trigger*> shall not appear with the same <*path*> in another <*input*> for the same state.

Model

An <*asterisk save list*> is transformed to a list of <*save*> items containing the complete valid input signal set of the enclosing agent, except for any <*type identifier*> of an implicit input signal (for a priority trigger, a continuous transition or a guard) or any <*type identifier*> contained in any other <*trigger*> and <*saved trigger*> of the <*transition*>.

NOTE – If the <*asterisk save list*> is applied to a substate within a composite state, the substate has an input or save for every receivable signal, therefore in this substate it is not possible to trigger a transition on a composite state that uses the composite state.

A <*save*> with virtuality is a virtual save and may be redefined. The <*save*> in the redefinition replaces the <*save*> in the supertype of the containing <*agent definition*>.

Mapping

A <*save*> represents deferrableTrigger of the sourceState. Each <*type identifier*> item in a <*stimulus*> of a <*saved trigger*> represents a Trigger. The event of trigger is a SignalEvent or TimeEvent, depending on whether the <*type identifier*> references a signal or a timer, respectively. The <*path*> (if present in a <*stimulus*>) represents port. An <*asterisk save list*> represents an AnyReceiveEvent as trigger.

If the <*save*> redefines an <*input*>, <*save*> or priority input in a supertype, the redefined <*input*>, <*save*> or priority input, respectively, represents redefinedTransition.

I.7.3.7 Spontaneous transition

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

Concrete grammar

<*spontaneous transition*> ::=
    `input` {`none` | `( none )` } [<*guard*>] <*stereotype*>* <*transition statements*>

A virtual spontaneous transition may be redefined to a spontaneous transition. A state shall not have more than one virtual spontaneous transition.

Model

A <*spontaneous transition*> with virtuality is a virtual spontaneous transition and may be redefined. The <*spontaneous transition*> in the redefinition replaces the <*spontaneous transition*> in the supertype of the containing <*agent definition*>.

Mapping

A <*spontaneous transition*> represents a Transition. The keyword **`none`** represents trigger; the event of the trigger is a SignalEvent. The <*expression*> in <*guard*> (if present) represents guard; the <*transition statements*> represent the effect.

If the <*spontaneous transition*> redefines a <*spontaneous transition*> in a supertype, the redefined <*spontaneous transition*> represents redefinedTransition.

I.7.3.8 Implicit transition

Any signal not handled by an explicit input or save is consumed by an implicit transition without a change of state.

Model

The set of stimuli contained in the triggers, priority inputs and the saved triggers of the transition (explicitly or via asterisk input list or asterisk save list) after applying above transformations is referred to as the local signal set. If the local signal set is the same as the complete valid input signal set of the agent, there are no implicit transitions; otherwise an implicit transition is constructed for each stimulus that is not in the local signal set. Each implicit transition is an <*input*> with the stimulus as <*trigger*> and <*transition statements*> that contain a single <*nextstate statement*> leading back to the same state.

A stimulus is in the local signal set of a composite state if it is in the local signal set for a transition of the composite state, or in the local signal set of any enclosing composite state.

I.7.3.9 Labelled transition

A labelled transition is not in itself a complete transition. Rather, it provides the continuation of another transition.

Concrete grammar

<*labeled transition*> ::=
    <*label*> <*transition statements*>

A virtual labelled transition may be redefined to a labelled transition.

All the <*label*> items defined in the <*transition statements*> of a service shall be distinct.

A <*transition terminating statement*> in the <*transition statements*> may reference only a <*label*> in the same service.

A <*transition terminating statement*> in the <*transition statements*> of a service of a specialized agent may reference a <*label*> defined in the supertype for the same service.

Mapping

A <*labeled transition*> represents a Transition with empty trigger and empty guard. The <*label*> represents name; the <*transition statements*> represent the effect.

I.7.4 Substate machine

A substate machine is a mechanism to nest one state machine inside another. That is, a state machine can be associated with a given state which is referred to as a composite state. The interpretation of a substate machine is started when the associated composite state is entered, and it is exited when one of the exit points of the substate machine is reached.

I.7.4.1 Composite state

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

The properties of a composite state are defined by a composite state definition together with transitions defined for the composite state. The latter transitions apply to all the substates of the composite state.

Concrete grammar

<*composite state definition*> ::=
    `state` <*name*> <*state signature*> <*stereotype*>* <*entities*>

An <*entity*> in the <*entities*> of a <*composite state definition*> shall be one of the following: <*state definition*>, <*transition*>, <*start transition*>, <*labeled transition*>, <*entry action*>, <*exit action*> or <*variable definition*>.

An <*entity*> in a <*composite state definition*> shall not contain a <*template*>.

A redefined composite state definition shall contain at least the <*state connection points*> of the virtual composite state definition it redefines.

<*state signature*> ::=
    [<*state connection points*>]

<*state connection points*> ::=
    <*inbound connection points*> [<*outbound connection points*>]
    | <*outbound connection points*> [<*inbound connection points*>]

<*inbound connection points*> ::=
    `in` { [<*name*>+[| **`,`**]] | **`(`** [<*name*>+[| **`,`**]] **`)`** }

<*outbound connection points*> ::=
    `out` { [<*name*>+[| **`,`**]] | **`(`** [<*name*>+[| **`,`**]] **`)`** }

The <*state connection points*> specify entry to and exit from a composite state. The <*inbound connection points*> specify start transitions that can be referenced from guards in a containing state machine. The <*outbound connection points*> specify return states that can be referenced from a <*return statement*> in a contained state machine.

A <*return*> with a <*name*> (a labelled return) shall reference only a <*name*> in the <*outbound connection points*> of the closest containing <*composite state definition*>.

NOTE – It is permitted for the same <*name*> to occur more than once in outbound connection points, but this has the same meaning as a single occurrence.

<*entry action*> ::=
    `entry` <*stereotype*>* <*statements*>

<*exit action*> ::=

  exit <*stereotype*>* <*statements*>

Model

An <*entry action*> or <*exit action*> that contains virtuality is a virtual entry action or virtual exit action, respectively, and may be redefined. The <*entry action*> or <*exit action*> in the redefinition replaces the <*entry action*> or <*exit action*>, respectively, in the supertype of the containing <*agent definition*>.

Mapping

A <*composite state definition*> represents State. The <*name*> represents name, <*entities*> represents Region. If the <*composite state definition*> redefines a <*state definition*> in a supertype, the redefined <*state definition*> represents redefinedClassifier.

A name in <*inbound connection points*> represents an entry Pseudostate. A name in <*outbound connection points*> represents an exit Pseudostate.

The <*entry action*> and <*exit action*> represent a StateMachine with exactly one Region with a single Transition. The source of the Transition is a Pseudostate with kind initial; the target is a FinalState; the <*statements*> represent effect. The entry and exit property, respectively, shall refer to this StateMachine.

I.7.4.2 Connect transition

Connect transitions are performed when a composite state is exited.

Concrete grammar

<*connect transition*> ::=

  [<*label*> | <*asterisk connect list*>] <*transition statements*>

<*asterisk connect list*> ::=

  [***** [**(** <*name*>+[| **,**] **)**] **]**

A <*connect transition*> shall appear only in a <*transition*> containing a <*name*> in its <*state list*> that references a <*composite state definition*>.

The <*label*> shall contain only <*name*> items in <*outbound connection points*> (denoting transitions taken when exiting a composite state via a labelled return) or **default** (indicating a transition taken when an unlabelled return is performed in a composite state).

A virtual connect transition may be redefined to a connect transition.

Model

A <*connect transition*> that contains an <*asterisk connect list*> is transformed into a <*label*> with a <*name*> for each <*name*> in the <*outbound connection points*> of the <*composite state definition*> in question, including **default** for the unlabelled <*return*>, except those <*name*> items mentioned in the <*asterisk connect list*>.

A <*connect transition*> with virtuality is a virtual connect transition and may be redefined. The <*connect transition*> in the redefinition replaces the <*connect transition*> in the supertype of the containing <*agent definition*>.

Mapping

A <*connect transition*> represents a Transition with empty trigger and empty guard. The <*name*> in <*label*> represents name; the <*transition statements*> represent the effect.

I.7.5 Transition statements

A transition consists of a sequence of statements to be interpreted by the agent.

Concrete grammar

<*transition statements*> ::=
    <*statements*>

Model

For a <*transition statements*> item, an <*operation definition*> with anonymous <*name*> and empty parameters is constructed. The <*statement*> items in the <*transition statements*> form the <*operation statements*>.

If at least one <*statement*> in the <*statement*> items contains a <*transition terminating statement*>, the <*type*> in <*result*> is the predefined type `Integer`. The <*statement*> items of the <*transition statements*> are removed and a <*decision statement*> is inserted in their stead with a call to the anonymous operation as <*expression*>. Every <*transition terminating statement*> of the <*transition statements*> is replaced in the <*operation definition*> by a <*return statement*>, returning a different `Integer` and a <*branch*> is inserted into the <*decision statement*>, with that `Integer` as <*constraint*> and the corresponding <*transition terminating statement*> as the single <*statement*> in its <*statements*>.

Otherwise, there is no <*result*>. The <*statement*> items are removed and a call to the anonymous operation is inserted in their stead.

Mapping

The <*transition statements*> represent an Activity containing a single node, which is either a ConditionalNode or a CallOperationAction, as created by the Model.

I.8 Sequential behaviour

For a system to perform some useful activity, the transitions of the state machine of agents must either have some effect on the external environment or have some effect on the internal state of the system.

Examples of such effects might be the calling of an operation, the sending of a signal to the environment or the updating of an internal value. The details of these actions typically depend on internal values of the system and may be performed through a series of computational steps.

At a certain level, the specification of such computational steps may seem similar to programs in a general-purpose programming language, such as C or Java. However, a specification language aims to describe such computations at an appropriate level of abstractions and to separate the specification of the computation from its implementation. For example, the details of the implementation of sending a message between agents should be hidden from the specification. As another example, the allocation and management of memory should not be of concern to the specification.

This clause describes the general actions related to flow of control, sending of messages and timers. Behaviour associated with specific types is described in clause I.9.

I.8.1 Statements

Statements are a sequence of actions, such as sending a message, calling an operation, assigning a data item to a variable, etc., interpreted in the order they occur.

Concrete grammar

<*statements*> ::=
  { <*stereotype*>* { <*statement*> <*stereotype*>* }* **}**
  | <*empty statement*>

<*statement*> ::=
  <*variable definition*>
  | <*expression statement*>
  | <*labeled statement*>
  | <*compound statement*>
  | <*conditional statement*>
  | <*loop statement*>
  | <*action statement*>
  | <*terminating statement*>
  | <*transition terminating statement*>
  | <*selected statements*>

If a <*variable definition*> is present, a <*name*> in the <*variable definition*> shall not be referenced, directly or indirectly, in a <*variable access*> in the containing <*statements*> before the <*variable definition*>.

<*terminating statement*> ::=
  <*return statement*>
  | <*break statement*>
  | <*continue statement*>

<*transition terminating statement*> ::=
  <*nextstate statement*>
  | <*stop statement*>
  | <*goto statement*>

<*action statement*> ::=
  { <*output*> | <*set*> | <*reset*> | <*assignment*> } **;**

<*conditional statement*> ::=
  <*decision statement*> | <*type decision statement*>
  | <*if statement*>

<*loop statement*> ::=
  <*while statement*>
  | <*for statement*>

<*compound statement*> ::=
  <*statements*>

<*labeled statement*> ::=
  <*label*> <*statement*>

<*label*> ::=
  [[<*name*>+[| **,**]] **]**

NOTE – To avoid awkward wording, when a <*statement*> is the <*statement*> of a <*labeled statement*>, the <*label*> will be referred to as the "label of the <*statement*>".

<*expression statement*> ::=
  <*expression*> **;**

The <*expression*> in <*expression statement*> shall not be a <*type coercion expression*>.

<*empty statement*> ::=

  ;

Model

If a <*default*> is present in a <*variable definition*>, for every <*name*> in the <*variable definition*>, an <*assignment*> is created from the <*expression*> of the <*default*> to the <*name*> as <*location*> and inserted into the <*statements*> containing the <*variable definition*> immediately following the <*variable definition*>. The <*default*> is then removed from the <*variable definition*>.

An <*empty statement*> is removed from the containing <*statements*>.

For a <*terminating statement*> which is not the last <*statement*> in the containing <*statements*>, a <*compound statement*> is constructed such that the <*terminating statement*> is its only <*statement*> and is used instead of the <*terminating statement*>.

Mapping

The <*statements*> represent a SequenceNode. Each <*statement*> in its list of <*statement*> items represents an executableNode, in the order of occurrence.

Each <*statement*> represents an Action or an ActivityNode.

The <*name*> of the <*label*> represents the name of the ActivityNode represented by the <*statement*>.

An <*expression statement*> represents an <<ExpressionAction>>ValueSpecificationAction. The <*expression*> represents the value.

A <*compound statement*> represents a SequenceNode. The <*statements*> represent executableNode. If the <*compound statement*> is not a <*labeled statement*>, an anonymous <*name*> is created as name. The <*variable definition*> items contained in <*statements*> are collected in a list where each represents a Variable in variable (see clause I.8.1).

I.8.2 Terminating statements

Terminating statements end the interpretation of the current transition, operation or statement sequence. The stop statement also terminates the interpretation of the containing agent instance.

I.8.2.1 Nextstate

A transition can end by entering a state, either a different state than the original one or the same state.

Concrete grammar

<*nextstate statement*> ::=

  nextstate { <*named nextstate*> | <*dash nextstate*> | <*history nextstate*> }**;**

<*named nextstate*> ::=

  <*unqualified name*> [<*entry path*>]

The <*unqualified name*> of a <*named nextstate*> shall reference a <*name*> contained in the <*state list*> of a <*transition*> in the same service.

<*entry path*> ::=

  via <*name*>

The <*name*> in the <*entry path*> shall be a <*name*> in the <*inbound connection points*> of the <*composite state definition*> referenced by the <*unqualified name*> of the <*named nextstate*>.

<*dash nextstate*> ::=

  -

<*history nextstate*> ::=
    --

A <*dash nextstate*> or <*history nextstate*> shall not be a <*transition terminating statement*> of a <*start transition*>.

Mapping

A <*nextstate statement*> with a <*named nextstate*> without an <*entry path*> represents the target property of the containing Transition and references the State with the <*unqualified name*> as name.

A <*nextstate statement*> with a <*named nextstate*> containing an <*entry path*> represents the target property of the containing Transition and references a Pseudostate with kind entryPoint where the <*unqualified name*> represents the name of the State that owns this Pseudostate and the <*name*> in the <*entry path*> represents the Pseudostate name.

A <*nextstate statement*> with a <*dash nextstate*> represents the target property of the containing Transition and references a Pseudostate with kind shallowHistory.

A <*nextstate statement*> with a <*history nextstate*> represents the target property of the containing Transition and references a Pseudostate with kind deepHistory.

I.8.2.2 Stop

An agent ceases to exist after it interprets a stop statement.

Concrete grammar

<*stop statement*> ::=
    stop ;

Model

For every <*agentset definition*> of the containing <*agent definition*>, a <*loop statement*> is inserted before the <*stop statement*> which iterates over the agentset and performs an <*output*> with **stop** as <*expression*> and the agent instance as <*destination*> for every active agent instance in the agentset.

Mapping

A <*stop statement*> represents a <<Stop>>ActivityFinalNode.

I.8.2.3 Return

The interpretation of an operation is terminated by a return statement.

Concrete grammar

<*return statement*> ::=
    <*return*> **;**

<*return*> ::=
    return { [<*expression*>] | <*exit path*> }

A <*return*> with an <*expression*> shall be used only within an <*operation definition*>. The <*expression*> shall not be omitted unless the <*result*> is also omitted. A <*return*> in a <*constructor definition*> shall not have an <*expression*>.

The <*expression*> of a <*return*> shall be type compatible with the <*type*> of the <*result*> of the enclosing <*operation definition*>.

<*exit path*> ::=
    via <*name*>

A <*return*> with an <*exit path*> shall be directly contained in a <*composite state definition*> that contains in its <*outbound connection points*> a <*name*> item with the <*name*> of the <*exit path*>.

Model

A <*return statement*> without an <*exit path*> contained (directly or indirectly) within a <*composite state definition*> is replaced by a <*return statement*> with **`default`** as the <*name*> of the <*exit path*>.

Mapping

A <*return statement*> with an <*exit path*> represents a FinalState. The <*name*> in the <*exit path*> represents name which references an exit connection point of the containing state machine. Otherwise, a <*return statement*> represents a <<`Return`>>ActivityFinalNode. If an <*expression*> is present, it represents value.

I.8.2.4 Goto, Break and Continue

A break statement resumes interpretation following a designated statement. A continue statement resumes interpretation at the designated statement. A goto statement identifies a labelled transition where interpretation resumes.

Concrete grammar

<*goto statement*> ::=
    `goto` <*name*> **`;`**

The <*name*> of a <*goto statement*> shall be a <*name*> in a <*label*> of a <*labeled transition*>.

<*break statement*> ::=
    `break` [<*name*>] **`;`**

The <*name*> of a <*break statement*> shall be the <*name*> contained in a <*label*> of a <*labeled statement*>.

A <*break statement*> shall be directly or indirectly contained within a <*labeled statement*> such that the <*name*> is the <*name*> of the <*label*>, or within a <*loop statement*>, if a <*name*> is not present.

<*continue statement*> ::=
    `continue` [<*name*>] **`;`**

The <*name*> of a <*continue statement*> shall be the <*name*> contained in a <*label*> of a <*labeled statement*>.

A <*continue statement*> shall be directly or indirectly contained within a <*labeled statement*> such that the <*name*> is the <*name*> of the <*label*>, or within a <*loop statement*>, if a <*name*> is not present.

Mapping

A <*goto statement*> that is a <*transition terminating statement*> represents a Pseudostate with kind equal to junction. The <*name*> represents name.

A <*break statement*> represents a <<`Break`>>OpaqueAction. The name, if present, represents name.

A <*continue statement*> represents a <<`Continue`>>OpaqueAction. The name, if present, represents name.

I.8.3 Action statements

An action statement may cause one or more communications to happen. These communications can trigger a state machine transition (output signals or timers).

I.8.3.1 Output

An output statement causes the sending of one or more signals.

Concrete grammar

<*output*> ::=
  `output` <*output clause*>+[| `,`]

<*output clause*> ::=
  <*expression*> [<*priority*>] [<*communication constraints*>]

The type of the <*expression*> in the <*output clause*> shall reference a type definition which has been referenced in a <*stimulus definition item*> of a <*signal definition*> or the <*expression*> shall be a `stop`.

A <*signal definition*> referenced by the type of the <*expression*> in <*output clause*> shall be conveyed by an <*outbound port constraint*> of a <*port definition*> contained in the <*agent definition*> that contains the <*output clause*>.

The <*expression*> in <*priority*> shall be the predefined type `Natural`.

NOTE 1 – If <*priority*> is omitted, the signal has the highest signal priority.

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

<*destination*> ::=
  `to` <*primary*>

The static type of the <*primary*> in <*destination*> shall either be the predefined type `Agent` (see clause I.5.1), or a subtype thereof.

<*path*> ::=
  `via` { <*channel destination*> | <*identifier*> }

The <*channel destination*> in <*path*> shall not be a <*service destination*>.

NOTE 2 – Consequentially, a <*channel destination*> in a path references a <*port definition*>, either as an <*agent port*> or an <agentset *port*>.

For each <*path*> with <*channel destination*>, the <*signal definition*> referenced by the type of the <*expression*> in the <*output clause*> shall be conveyed by the <*outbound port constraint*> of the referenced port definition.

The <*identifier*> in <*path*> shall reference a <*channel definition*> reachable from the enclosing <*agent definition*>, a <*port definition*> or the sole <*signallist definition*> in a <*port definition*> without a <*name*>.

For each <*path*> with <*identifier*>, the <*signal definition*> referenced by the type of the <*expression*> in <*output clause*> shall be conveyed by either the referenced <*channel definition*> or the <*port definition*> referenced in either <*channel destination*>.

Model

If the <*name*> in a <*path*> in <*communication constraints*> references a <*channel definition*> and the <*target channel end*> references an <*agentset definition*> of the <*agent definition*> containing the <*output clause*> in its <agentset *port*>, then this is replaced by a <*path*> referencing that <agentset *port*>.

If the <*name*> in a <*path*> in <*communication constraints*> references a <*channel definition*> and the <*target channel end*> does not contain an <agentset *port*>, then this is replaced by a <*primary*> referencing a <*port definition*> contained in the <*agent definition*> containing the <*output*> which is referenced in a <*channel destination*> of the <*source channel end*>.

If there is more than one <*output clause*> specified in an <*output*>, the <*output*> is transformed to a sequence of <*output*> items, each with a single <*output clause*> in the same order as specified in the original <*output*>. The <*communication constraints*> are repeated in each <*output clause*>.

If the <*communication constraints*> of an <*output clause*> contain more than one <*destination*>, this is shorthand for replacing the <*output*> by a sequence of <*output*> items, one for each <*destination*>. Each <*output*> has the same original <*output clause*>, except that in each case the <*communication constraints*> contain only one <*destination*> taken in order from the original <*communication constraints*>.

For the <*expression*> in an <*output clause*>, a <*variable definition*> is created for a temporary variable with an anonymous <*name*>, the type of <*expression*> as <*type*> and the <*expression*> as <*default*>. The <*variable definition*> is inserted before the containing <*output*>. A reference to the anonymous variable is inserted in place of the <*expression*> in the <*output clause*>.

Mapping

Each <*output clause*> in an <*output*> represents a SendSignalAction.

The <*name*> of the <*type*> of the variable referenced in <*expression*> (see above Model) represents the qualifiedName of the signal. If the <*communication constraints*> contain a <*destination*>, the <*primary*> represents target. If the <*communication constraints*> contain a <*path*>, the <*channel destination*> or <*identifier*> represents onPort. Each attribute of the <*type*> of the variable referenced in <*expression*> represents an argument.

NOTE 3 – This Recommendation does not provide a mapping for <*priority*> in an <*output clause*> to SDL and therefore all signals will be output without priority.

I.8.3.2 Timer

Timers cause a stimulus to occur at some future specified time. Timers, once defined, may be set or reset. When a set timer expires, an associated stimulus is put on the input queue of the containing agent.

Concrete grammar

<*timer definition*> ::=
    [<*template*>] **timer** <*stimulus definition item*>+[| **,**] [<*default*>] **;**

The syntactic constraints in clause I.6.3 for <*stimulus definition item*> shall apply.

The <*expression*> in the <*default*> shall be a constant of the predefined type **Duration**.

A <*context parameter*> in the <*template*> of a <*timer definition*> shall be a <*type context parameter*>, <*variable context parameter*> or <*value context parameter*>.

<*set*> ::=
    set <*timer set clause*>+[| **,**]

<*timer set clause*> ::=
    <*type*> [<*actual parameters*>] [<*default*>]

The <*type*> of a <*timer set clause*> shall reference a <*class definition*> referenced by a <*stimulus definition item*> that is contained in a <*timer definition*>.

The <*default*> in a <*timer set clause*> shall be omitted only if the referenced <*timer definition*> has a <*default*>. The <*expression*> in the default shall be an <*operand*> of the predefined type **Duration**.

<*reset*> ::=
    reset <*timer reset clause*>+[| **,**]

<*timer reset clause*> ::=
    <*type*> [<*actual parameters*>]

The <*type*> of a <*timer reset clause*> shall reference a <*class definition*> referenced by a <*stimulus definition item*> that is contained in a <*timer definition*>.

Each <*type*> of an <*actual parameter*> in <*actual parameters*> in a <*timer set clause*> or <*timer reset clause*> shall correspond by position to the <*type*> in the <*parameter*> of the <*parameters*> of the referenced <*timer definition*>.

Model

When more than one <*stimulus definition item*> occurs in a <*timer definition*>, a separate <*timer definition*> is created for each <*stimulus definition item*> and the original <*stimulus definition item*> is deleted from the <*timer definition*>.

A <*stimulus definition item*> with <*parameters*> is transformed as defined in clause I.6.3.

If a <*timer set clause*> has no <*default*>, a <*default*> is constructed from `now` added to the <*default*> of the referenced <*timer definition*>.

If a <*set*> contains several <*timer set clause*> items this is shorthand notation for specifying a sequence of <*set*> items, one for each <*timer set clause*> such that the original order in which they were specified is retained.

If a <*reset*> contains several <*timer reset clause*> items this is shorthand notation for specifying a sequence of <*reset*> items, one for each <*timer reset clause*> such that the original order in which they were specified is retained.

Mapping

Each <*stimulus definition item*> in <*timer definition*> represents a <<`Timer`>> Signal. The <*type identifier*> of that <*stimulus definition item*> represents name; each <*parameter*> in <*parameters*> represents ownedAttribute. The <*default*> corresponds to defaultValue.

NOTE – The interpretation of a timer definition with a <*template*> is given in clause I.4.4.2.

Each <*timer set clause*> in a <*set*> represents a <<`SetAction`>>SendSignalAction. The <*type*> represents signal. The <*actual parameters*>, if present, represent the argument list. The <*default*> item represents timeExpression.

Each <*timer reset clause*> in a <*reset*> represents a <<`ResetAction`>>SendSignalAction. The <*type*> represents signal. The <*actual parameters*>, if present, represent the argument list.

I.8.4 Loop statement

A loop statement is a mechanism to perform repeated actions, either iterating over the elements of a collection, for a defined number of times, or until some condition is met.

Concrete grammar

<*for statement*> ::=
    `for` { <*loop clause*>+[| **`,`**] [<*loop test*>] | **`(`** <*loop clause*>+[| **`,`**] [<*loop test*>] **`)`** }
        <*statements*> [<*loop finalization*>]

<*loop clause*> ::=
    <*loop variable*> [<*loop step*>]

<*loop variable*> ::=
    <*loop variable definition*> | <*loop variable use*>

The <*name*> of a <*loop variable*> shall not appear as the <*location*> of an <*assignment*> in the <*statements*> of a <*for statement*>.

<*loop variable use*> ::=
    <*name*> <*stereotype*>*

A <*name*> of a <*loop variable use*> shall not identify a variable defined in a <*loop variable definition*> of the same <*for statement*>.

<*loop variable definition*> ::=
    [<*modifier*>] <*name*> **:** <*type*> <*stereotype*>*

<*loop step*> ::=
    in <*explicit enumeration*>
  | **in** <*collection enumeration*>
  | <*iteration*>

<*explicit enumeration*> ::=
    <*enumeration*>

If the <*loop step*> is an <*explicit enumeration*>, each element of the <*enumeration*> shall be type compatible with the <*type*> of the <*loop variable*>.

<*collection enumeration*> ::=
    <*expression*>

If the <*loop step*> is a <*collection enumeration*>, the <*expression*> shall be a collection for which each element is type compatible with the <*type*> of the <*loop variable*> or it shall reference an <*agentset definition*>.

<*iteration*> ::=
    := <*iteration step*> [**from** <*iteration start*>] [**to** <*iteration end*>]

<*iteration step*> ::=
    <*expression*>

<*iteration start*> ::=
    <*expression*>

<*iteration end*> ::=
    <*expression*>

If the <*loop step*> is an <*iteration*>, the <*expression*> in <*iteration step*> shall be a call to an operation with a return type that is type compatible with the <*type*> of the <*loop variable*>. The type of <*iteration end*> shall be the predefined type **Boolean**.

<*loop test*> ::=
    while <*expression*>

The <*expression*> of a <*loop test*> shall have the predefined **Boolean** type.

<*loop finalization*> ::=
    then <*statements*>

<*while statement*> ::=
    while { <*expression*> | **(** <*expression*> **)** } <*statements*>

The <*expression*> of a <*while statement*> shall have the predefined **Boolean** type.

Model

If the <*loop step*> is an <*explicit enumeration*>, an operation definition with an anonymous <*name*> and a result <*type*> that is the static type of a <*variable access*> to the corresponding <*loop variable*> is constructed such that it returns each subsequent element in the <*enumeration*> (see clause I.9.1.8) each time it is called, until the range condition is exhausted.

If the <*loop step*> is a <*collection enumeration*>, an operation definition with an anonymous <*name*> and a result <*type*> that is the static type of a <*variable access*> to the corresponding <*loop variable*> is constructed such that it returns each subsequent element in the collection returned by the <*expression*> each time it is called, until the collection is exhausted.

If the <*loop step*> is an <*iteration*>, an operation definition with an anonymous <*name*> and a result <*type*> that is the static type of a <*variable access*> to the corresponding <*loop variable*> is constructed such that it first, in the first iteration only, interprets the <*iteration start*> expression, if present. The <*expression*> in <*iteration end*> is interpreted and if it returns the **`Boolean`** value **`false`**, the data items provided by this loop clause are exhausted. Otherwise, the <*iteration step*> is interpreted and the data item returned is returned by the <*loop step*>.

Mapping

A <*for statement*> represents a LoopNode. For each <*loop variable*> in the <*loop clause*> list, if the <*loop variable*> is a <*loop variable definition*>, it represents a Variable in variable (see clause I.8.1). The <*stereotype*> list then applies to the Variable. The optional <*loop test*> is transformed into an evaluation of the <*loop test*>, and if the <*loop test*> evaluates to true, true is returned; otherwise, the <*statements*> in <*loop finalization*> are interpreted and false is returned. The so transformed <*loop test*> represents a SequenceNode that is the testPart. The isTestedFirst attribute is true. The <*expression*> of the <*loop test*> represents the corresponding ExpressionAction. The <*statements*> item represents a SequenceNode that is the bodyPart. For each <*loop variable*>, a sequence of AddVariableValueActions, with the <*loop variable*> representing variable and a call to the operation constructed in the Model for the corresponding <*loop step*> representing value, is constructed and represents the stepGraphPart. The name is an anonymous name, unless the <*for statement*> was the statement of a <*labeled statement*> (in which case the <*name*> of the <*labeled statement*> represents name).

NOTE – This Recommendation does not provide a mapping of finalization to SDL and thus the <loop *finalization*> is interpreted as part of the <*loop test*>.

A <*while statement*> represents a LoopNode. The <*expression*> represents an ExpressionAction that is the sole action in a SequenceNode that is the testPart. The isTestedFirst attribute is true. The <*statements*> represents a SequenceNode that is the bodyPart. The setupPart and stepGraphPart are empty. The name is an anonymous name, unless the <*while statement*> was the statement of a <*labeled statement*> (in which case the <*name*> of the <*labeled statement*> represents name).

I.8.5 Conditional statements

A conditional statement allows one of a set of interpretation paths to be taken. The path chosen is based on the interpretation of a Boolean condition, on the interpretation of an expression and its matching against a set of conditions or on the matching of the type of an expression.

I.8.5.1 Decision statement

In a decision statement, an expression is interpreted and the branch whose constraint contains the result of the expression is interpreted. If there is no match and an else branch exists, the else branch is interpreted. If there is no match and an else branch does not exist, interpretation continues after the decision statement.

Concrete grammar

<*decision statement*> ::=
    `switch`{ <*expression*> | **`(`** <*expression*> **`)`** } <*branches*> [<*else branch*>]

<*branches*> ::=
    `{` <*branch*>* **`}`**

<*branch*> ::=
    <*constraint*> <*statements*>

The <*constraint*> shall be a <*range constraint*>, that is, each <*range*> shall be type compatible with the <*type*> of the <*expression*> in <*decision statement*>.

<*else branch*> ::=
    `else` <*statements*>

Mapping

A <*decision statement*> represents a ConditionalNode. Each <*branch*> item in the <*branch*> list of the <*branches*> represents a Clause in clause. The <*statements*> represent a SequenceNode that is the body of that Clause, and the test of the Clause is an ExpressionAction with value represented by an SdlExpression that is a RangeCheckExpression where <*expression*> maps to expression and <*type*> maps to the rangeCondition.

If there is an <*else branch*>, the <*statements*> represent a SequenceNode that is the body of a Clause, with the test being an ExpressionAction with value represented by true. The successorClause set is empty. This Clause is the sole member of the successorClause set of all other Clause items.

If there is no <*else branch*>, the ConditionalNode contains Clause items only for each <*branch*> item in the <*branch*> list of the <*branches*>. The successorClause set of all Clause items is empty.

I.8.5.2 If statement

In an if statement, a Boolean expression is interpreted and if it returns the predefined `Boolean` value `true`, the statements are interpreted; otherwise, the else branch, if present, is interpreted.

Concrete grammar

<*if statement*> ::=
    `if` { <*expression*> | `(` <*expression*> `)` } <*statements*> [<*else branch*>]

The <*expression*> shall be the predefined type `Boolean`.

Model

An <*if statement*> is transformed into a decision statement with <*expression*> as the <*expression*>, a <*branch*> with <*statements*>, a <*constraint*> formed from the literal `true` and the <*else branch*>, if present.

Mapping

An <*if statement*> represents a ConditionalNode. The <*statements*> represent a SequenceNode that is the body of the first Clause in clause. The test of this Clause is an ExpressionAction with value represented by "<*expression*> `== true`".

If there is an <*else branch*>, the <*statements*> represent a SequenceNode that is the body of a Clause in clause. The test of this clause is an ExpressionAction with value represented by true. This Clause is added to clause as a successor to all other Clause items.

If there is no <*else branch*> there is no additional Clause in clause.

I.8.5.3 Type decision statement

In a type decision statement, an expression is interpreted and the branch that matches the type of the expression is interpreted. Overlapping constraints are not allowed. If there is no match and an else branch exists, the else branch is interpreted. If there is no match and an else branch does not exist, interpretation continues after the type decision statement.

<*type decision statement*> ::=
    `switch` { <*expression*> **`:?`** | **`(`**<*expression*> **`:? )`** } <*type branches*> [<*else branch*>]

<*type branches*> ::=
    `{` <*type branch*>* **`}`**

<*type branch*> ::=
    `[` <*type*>+[| **`,`**] **`]`** <*statements*>

Model

If a <*type branch*> contains more than one <*type*> in its <*type*> list, for each <*type*> a new <*type branch*> is constructed from that <*type*> and the <*statements*> of the original <*type branch*>. The original <*type branch*> is then deleted.

Mapping

A <*type decision statement*> represents a ConditionalNode. Each <*type branch*> item in the <*type branch*> list of <*type branches*> represents a Clause. The <*statements*> represent a SequenceNode that is the body of that clause, and the test of the clause is an ExpressionAction with value represented by an SdlExpression that is a RangeCheckExpression where <*expression*> maps to expression and <*type*> maps to parentSortIdentifier. If <*type*> is a <*constrained type*>, its <*constraint*> represents rangeCondition.

If there is an <*else branch*>, the <*statements*> represent a SequenceNode that is the body of a Clause, with the test being an ExpressionAction with the value represented by the Boolean SdlExpression value true. This Clause is added to clause as a successor to all other Clause items.

If there is no <*else branch*>, the ConditionalNode contains Clause items only for each <*type branch*> item in the <*type branch*> list of the <*type branches*>.

I.9 Data

This clause defines the concept of data in a specification. This includes the data terminology, the concepts to define new data types and the predefined data types.

Data definitions are principally concerned with the specification of (data) types. A (data) type defines a set of elements or data items and a set of operations that are allowed to be applied to these data items. The operations are the properties of (data) types. Data types are defined by class definitions. Interface definitions define abstract types.

A (data) type consists of a set of instances and one or more operations. As an example, the predefined type **`Boolean`** consists of the elements **`true`** and **`false`**. Among the operations of the type **`Boolean`** are `'=='` (equal), `'!='` (not equal), `'!'` (not), `'&&'` (and), `'||'` (or) and `'xor'`. As a further example, the predefined data type **`Integer`** consists of the elements `0`, `1`, `2`, etc., up to the largest **`Integer`**, the elements `-1`, `-2`, `-3`, etc., down to the smallest **`Integer`** and the operations `'=='`, `'!='`, `'+'`, `'-'`, `'*'`, `'/'`, `'mod'`, `'rem'`, `'<'`, `'>'`, `'<='` and `'>='`.

NOTE 1 – The predefined type **`Integer`** is conceptually infinite, but an implementation will have to limit the type to some finite range.

NOTE 2 – The language provides several predefined data types, which are familiar in both their behaviour and syntax. The predefined data types are described in [ITU-T Z.104].

The elements of a (data) type are either instances of the type or identities of agents. The elements of a type are defined either by:

a) explicitly enumerating the elements of the type (see clause I.9.1.4);

b) defining the set of properties that elements of the type shall satisfy (see clause I.9.1.2); or

c) as one of several types that are predefined.

Operations are defined as possibly taking elements of a type as parameters and possibly returning an element of a type. For instance, the application of the operation for summation (`'+'`) to two elements of the predefined **`Integer`** type is valid and returns an element of the predefined **`Integer`** type, whereas summation of elements of the predefined **`Boolean`** type is not.

Each data item belongs to exactly one type. That is, types never have data items in common. There are no implicit conversions between types. For example, a value of the predefined type **`Integer`** shall not be used in an operation where a parameter of the predefined type **`Real`** is required. If it is desired to use an **`Integer`** as an actual parameter to an operation which expects an actual parameter of type **`Real`**, an explicit conversion operation shall be applied, which converts the **`Integer`** value to a **`Real`** value, before the operation can be applied.

For some types, there may be literal forms to denote elements of the type: for example, for elements of the predefined type **`Integer`**, `2` is used to denote a particular instance, rather than constructing the instance from the operation `1 + 1`. It is allowed that more than one literal denote the same data item; for example, `12.0` and `12.000` denote the same **`Real`** data item. It is also allowed that the same literal is used for more than one type. Some types have no literals to denote the elements of that type, for example, the predefined **`Array`** type. In that case, the elements of such types are denoted by operations that construct the data item, possibly from elements of other types. Operations that construct elements of a type are referred to as *constructors*).

Variables are holders of data items and may have associated an element of a type. Elements of a type are associated with variables by assignment. When the variable is accessed, the associated data item is returned.

An expression denotes a data item. If an expression does not (directly or indirectly through operation calls) contain a variable or an imperative expression, e.g., if it is a literal of a given type, each occurrence of the expression will always denote the same data item.

An expression that contains (directly or indirectly through operation calls) variables or imperative expressions may have different results during the interpretation of a specification, depending on the data item associated with the variables.

I.9.1 Definition of data types

Class definitions (see clause I.9.1.1) and interface definitions (see clause I.9.1.2) are used to define (data) types. The creation of elements of the type is described in clause I.9.1.3 and clause I.9.1.4. Clause I.9.1.5 shows how to define the behaviour of the operations of a data type. The remaining subclauses detail a variety of properties related to data definitions including syntypes, attributes and constraints. Specialization (see clause I.4.3) allows the definition of a data type to be based on another data type, referred to as its supertype.

Since predefined data is defined in an implicitly used package **`Predefined`** (see clause I.1.7), the predefined types and their operations are available to be used throughout a specification.

I.9.1.1 Class

A <*class definition*> introduces a type that is visible in the enclosing scope unit. It may introduce a set of literals and/or operations.

The <*entities*> of the <*class definition*> define the properties of the specified type and how to construct sets of instances of this type.

Concrete grammar

<*class definition*> ::=
    [<*template*>] **`class`** <*name*> [<*specialization*>] <*stereotype*>* <*entities*>

At most one <*type identifier*> in the <*specialization*> shall refer to a <*class definition*>; all other <*type identifier*> items shall refer to <*interface definition*> items.

If <*specialization*> contains a <*type identifier*> referencing an <*interface definition*> and the stereotype <<`abstract`>> has not been applied to the <*class definition*>, the <*class definition*> shall provide <*operation definition*> items for all operations contained in the <*interface definition*>. If an operation with the same <*name*> and compatible signature is contained in several <*interface definition*> items referenced in the <*specialization*>, only one <*operation definition*> shall be provided.

An <*entity*> in the <*entities*> of a <*class definition*> shall be one of the following: <*class definition*>, <*interface definition*>, <*signallist definition*>, <*literal definition*>, <*syntype definition*>, <*attribute definition*>, <*variable definition*>, <*operation definition*>, <*constructor definition*>.

A <*class definition*> may be contained within another <*class definition*> or within an <*agent definition*>. The contained class may access visible entities of the enclosing class. A type definition contained within a <*class definition*> or within an <*agent definition*> shall not have the visibility <<`public`>> applied.

NOTE 1 – Consequentially, a contained <*class definition*> cannot be instantiated from outside the class or agent, that is, instances of the contained class shall be within instances of the enclosing class.

A <*constructor definition*> (see clause I.9.1.3) or <*literal definition*> (see clause I.9.1.4) describes how instances of the type defined by the <*class definition*> are constructed. An <*operation definition*> defines an operation for elements of the type (see clause I.9.1.5).

The <*name*> represents the type of the <*class definition*>, and this <*name*> also identifies the <*class definition*> as a scope unit in a <*qualifier*>.

A <*context parameter*> in the <*template*> of a <*class definition*> shall be a <*type context parameter*>, <*variable context parameter*>, <*operation context parameter*> or a <*value context parameter*>.

When a <*class definition*> has the stereotype <<`final`>> applied, the <*class definition*> shall not be further specialized.

Model

If no <*specialization*> is given, or if all <*type*> items in <*specialization*> are defined by <*interface definition*> items, a <*class definition*> implicitly specializes the predefined type **`Object`**, unless the stereotype <<`final`>> is present.

If no <*constructor definition*> is present, this is shorthand notation for an unnamed <*constructor definition*> with empty <*parameters*>, empty <*constructor initializer*> and absent <*operation statements*>.

A class definition with virtuality is a virtual class. A virtual class can be redefined to a subtype in specializations of the enclosing type, subject to virtuality constraints as further discussed in clause I.4.3.2. A redefined class replaces the virtual class it redefines.

Mapping

A <*class definition*> represents a <<`DataTypeDefinition`>>Class with the isActive property being false. The <*name*> represents name. Each type definition referenced by a <*type identifier*> in the <*specialization*>, if present, represents an element of general. For each <*entity*> in <*entities*>, an <*attribute definition*> represents an ownedAttribute; a <*literal definition*>, <*operation definition*> or <*constructor definition*> represents an ownedOperation; a <*class definition*> or <*syntype definition*> represents a nestedClassifier.

If the <*class definition*> redefines a <*class definition*> in a supertype, the redefined <*class definition*> represents redefinedClassifier.

NOTE 2 – The interpretation of a class definition with a <*template*> is given in clause I.4.4.2.

I.9.1.2 Interface

An <*interface definition*> defines an abstract type which identifies a set of operations that a subtype of this type shall define. These operations can be called on any concrete instance with a type compatible with the type defined by the interface.

An interface is used as the type of a variable or parameter. During interpretation an instance of a class that is type compatible with this interface shall be associated with this variable or parameter.

The defining context of the operations in an interface is the scope unit containing the interface. The operations in an interface are visible where the interface is visible.

Concrete grammar

<*interface definition*> ::=
    [<*template*>] **`interface`** <*name*> [<*specialization*>] <*stereotype*>*
    <*entities*>

The <*type identifier*> items in the <*specialization*> shall reference only <*interface definition*> items.

An <*entity*> in the <*entities*> of an <*interface definition*> shall be an <*attribute definition*> or <*operation definition*>. An <*operation definition*> in an <*interface definition*> shall contain only an <*empty statement*> in its <*operation statements*>. An <*entity*> in an <*interface definition*> shall not contain a <*template*>.

A <*context parameter*> in the <*template*> of an <*interface definition*> shall be a <*type context parameter*> or an <*operation context parameter*>.

Model

For each <*attribute definition*> in <*interface definition*>, the transformations in clause I.9.1.6 are applied. The <*entities*> are removed from the constructed <*operation definition*> items and these are inserted instead of the <*attribute definition*> into the <*entities*> of the <*interface definition*>.

An interface definition with virtuality is a virtual interface. A virtual interface can be redefined to a subtype in specializations of the enclosing type, subject to virtuality constraints as further discussed in clause I.4.3.2. A redefined interface replaces the virtual interface it redefines.

Mapping

An <*interface definition*> represents a <<`DataTypeDefinition`>>Class with the isAbstract property being true and with the isActive property being false. The <*name*> represents name. Each <*interface definition*> referenced by a <*type identifier*> in the <*specialization*>, if present, represents an element of general. Each <*operation definition*> in <*entities*> represents an ownedOperation.

If the <*interface definition*> redefines an <*interface definition*> in a supertype, the redefined <*interface definition*> represents redefinedClassifier.

NOTE – The interpretation of an interface definition with a <*template*> is given in clause I.4.4.2.

I.9.1.3 Constructor

Constructors specify a particular instance of a type. They either identify that instance or create a new instance possessing a specified set of properties.

Concrete grammar

<*constructor definition*> ::=
    <*ordinary constructor definition*> | <*default constructor definition*>

<*ordinary constructor definition*> ::=
    `constructor` [<*name*>] <*operation signature*> <*stereotype*>*
        <*constructor initializer*> [**`then`** <*operation statements*>]

The <*operation signature*> of a <*constructor definition*> shall not contain a <*result*>.

<*default constructor definition*> ::=
    `default` <*ordinary constructor definition*>

A type definition shall contain at most one <*default constructor definition*>. The <*default constructor definition*> shall not have <*parameters*> in its <*operation signature*>.

<*constructor initializer*> ::=
    <*operation statements*>

Each <*statement*> of the <*operation statements*> in the <*constructor initializer*> shall either be a <*constructor call*> to a constructor of the supertype, a <*constructor call*> to a different constructor of the containing type definition, an <*assignment*> to a variable resulting from an <*attribute definition*> or an <*operation call*> to a mutator defined in the containing type definition. A <*constructor call*> shall be the first <*statement*> of the <*operation statements*>, if present. The assignments to variables (either through <*assignment*> or through an <*operation call*> to a mutator) shall occur in the order of the corresponding <*attribute definition*> items, if any. Initialization of variables of supertypes shall be performed by invocation of a constructor of the supertype, not by assignments to the variables in the subtype.

NOTE 1 – If the constructor of the supertype has the same <*name*> as the constructor in the subtype, a <*qualified name*> needs to be used to identify the intended constructor.

Within the <*constructor initializer*>, it is allowed to give initial data items to the read-only variable and attributes defined as properties of the containing <*class definition*>.

Both the <*operation statements*> in the <*constructor initializer*> and the <*operation statements*> shall not contain calls to virtual operations. A <*constructor initializer*> shall not contain a <*return statement*>.

Model

If the <*name*> is omitted in a <*constructor definition*>, this is shorthand notation for a <*constructor definition*> with the <*name*> of the containing type definition.

If the containing type definition has a <*specialization*>, and there is no call to a constructor in the <*constructor initializer*>, a call to the constructor defined by a default constructor definition of the supertype that is not an interface, if present, is inserted as the first <*statement*> in <*constructor initializer*>; or otherwise a call to the constructor defined by the unnamed constructor definition without parameters of the supertype that is not an interface, if present, is inserted as the first <*statement*> in <*constructor initializer*>.

NOTE 2 – The transformation for <*default constructor definition*> is given in clause I.9.3.1.

Mapping

A <*constructor definition*> represents an <u>Operation</u> with <u>isOperator</u> <u>true</u> and is mapped as described further in clause I.9.1.5. The <u>Operation</u> has the stereotype <<`create`>> applied.

NOTE 3 – The interpretation of <*default constructor definition*> is given in clause I.9.3.1.

I.9.1.4 Literals

A literal definition specifies the contents of a type by enumerating the elements of the type. It may define operations that allow comparison between the elements of the type. These elements are called literals.

Concrete grammar

<*literal definition*> ::=
    `literals` <*literal definition item*>+[| **,**] <*stereotype*>* **;**

<*literal definition item*> ::=
    <*name*> [<*actual parameters*>] [<*entities*>]

If <*actual parameters*> are present in a <*literal definition item*>, the containing <*class definition*> shall contain a <*constructor definition*> without an <*identifier*> and with <*parameters*> that match the <*actual parameters*>.

Each <*actual parameter*> in <*actual parameters*> of a <*literal definition item*> shall be type compatible with the corresponding (by position) <*parameter*> of the matching unnamed <*constructor definition*>.

An <*entity*> in the <*entities*> of a <*literal definition item*> shall be an <*operation definition*> that defines a method.

A <*variable definition*> contained in a <*class definition*> that contains a <*literal definition*> shall have <*immutability*> **`const`**.

Model

For each <*literal definition item*> in a <*literal definition*>, an <*operation definition*> with an empty <*parameter*> list as <*parameters*> and the type specified by the enclosing <*class definition*> as the <*type*> in <*result*> is constructed. This operation has stereotypes <<`static`>> and <<`public`>>, unless a visibility stereotype was provided in the <*literal definition*> in which case that visibility stereotype is applied. If no <*actual parameters*> are given in a <*literal definition item*>, this operation returns the corresponding instance. Otherwise, if <*actual parameters*> are given, the operation returns the data item returned by the call to the (unnamed) constructor with <*parameters*> matching the <*actual parameters*>.

NOTE 1 – The instances corresponding to inherited literals in a specialized type consequentially are members of the specialized type, not the parent type that defined these literals.

NOTE 2 – The operations implicitly defined by the presence of <*actual parameters*> might, for example, introduce an ordering on the instances of the type specified by the containing <*class definition*>.

If a <*literal definition item*> contains <*operation definition*> items, then for each <*operation definition*>, an <*operation definition*> with matching <*name*>, <operation signature> and <*result*> is constructed such that the <*operation statements*> contain a <*decision statement*> where the <*operation statements*> of the original <*operation definition*> form the <*statements*> of a <*branch*> of the <*decision statement*> with the corresponding literal <*name*> as the label, unless an <*operation definition*> has already been constructed with matching <*name*>, <operation signature>, and <*result*>, in which case this branch is inserted into the <*decision statement*> of that <*operation definition*>. The variable **`this`** is used as the <*expression*> of the <*decision statement*>. The so constructed operations are inserted into the containing type definition. After this transformation, all <*operation definition*> items are removed from the <*literal definition item*>.

NOTE 3 – The operations defined in a <*literal definition item*> apply only to the literal defined in this <*literal definition item*>.

A <*literal definition*> with virtuality is a virtual literal definition. A virtual literal definition may be redefined in a specialization. A redefined literal definition is extended in the subtype. The literal definition item list of the redefined literal definition is the union of the inherited literal definition item list and the literal definition item list in the redefinition.

Mapping

NOTE 4 – The result of the Model for <*literal definition*> is mapped as the resulting <*operation definition*> items (see clause I.9.1.5).

I.9.1.5 Operation

Operations describe behaviour that can be invoked by an <*operation call*>. Operations may return data items as their return data item or through parameters. The behaviour of an operation is defined by its <*operation statements*>.

Concrete grammar

<*operation definition*> ::=
    [<*template*>] <*name*> <*operation signature*> <*stereotype*>* <*operation statements*>

An operation is identified within a scope unit by its <*operation signature*> and its <*name*>.

An <*operation definition*> in a <*class definition*> or <*interface definition*> with the stereotype <<`static`>> defines an operator. Otherwise, if the stereotype <<`static`>> is not applied to an <*operation definition*> in a <*class definition*> or <*interface definition*>, the <*operation definition*> defines a method. An <*operation definition*> not contained within a <*class definition*> or <*interface definition*> defines a procedure. The term "operation" refers to operators, methods and procedures.

When a virtual method is redefined, its <operation signature> shall be the same as the <operation signature> in the supertype, except that the <*type*> in the <*result*> of the redefined operation shall be type compatible to the <*type*> in the <*result*> of the supertype.

NOTE 1 – Because of this constraint it is not possible to define covariantly redefined methods.

A visibility stereotype shall not be applied to an <*operation definition*> that redefines an inherited operation.

An <*operation definition*> with the stereotype <<`abstract`>> applied defines an abstract operation. An abstract operation shall not be referenced in an <*operation call*>.

Overloading allows for more than one operation with the same <*name*> within the same scope. Operations can be overloaded as long as each <*operation definition*> has a different <*operation signature*>. If two operations in the same scope have the same <*name*> and have signatures that cannot be distinguished for each call to this operation, the behaviour of the specification is not defined. Two operation definitions with the same <*name*> items are allowed in an entity if they have a differing number of parameters, or they have the same number of parameters and contain at least one matching parameter with differing types.

A <*context parameter*> in the <*template*> of an <*operation definition*> shall be a <*type context parameter*>, <*operation context parameter*>, <*variable context parameter*> or <*value context parameter*>.

<*operation signature*> ::=
    <*parameters*> [`:` <*result*>]

A <*result*> shall be present in the <*operation signature*> of an <*operation definition*>.

<*parameters*> ::=
    `(` [<*parameter*>+[| `,`]] `)`

A <*list type*> shall be used only as the <*type*> of the last <*parameter*> in <*parameters*>. For a <*parameter*> with a <*list type*>, the <*name*> list shall contain only a single <*name*>. This <*name*> shall be used only as the <*expression*> that is the <*collection enumeration*> iterated over in a <*for statement*>.

<*parameter*> ::=
    {<*in parameter*> | <*out parameter*> | <*inout parameter*>} [<*default*>]

The <*expression*> in <*default*> shall be a constant of the <*type*> of the <*parameter*>.

<*in parameter*> ::=
  [`in`] [<*modifier*>] [<*name*>+[| `,`] `:`] <*type*> <*stereotype*>*

An <*in parameter*> is a read-only variable. The <*name*> of an <*in parameter*> shall not appear in a <*location*>.

<*out parameter*> ::=
  `out` [<*modifier*>] [<*name*>+[| `,`] `:`] <*type*> <*stereotype*>*

NOTE 2 – If an <*out parameter*> has `const` <*immutability*>, it can only be initialized in the <*operation signature*>.

<*inout parameter*> ::=
  `inout` [<*modifier*>] [<*name*>+[| `,`] `:`] <*type*> <*stereotype*>*

<*result*> ::=
  [<*modifier*>] <*type*>
  | `(` [<*modifier*>] <*type*> <*stereotype*>* `)`

NOTE 3 – If <*stereotype*> items need to be applied to the <*type*> of the <*result*>, parentheses need to be used.

<*operation statements*> ::=
  <*statements*>

A <*statement*> contained (directly or indirectly) in the <*operation statements*> of an <*operation definition*> contained in a <*class definition*> shall not be a <*transition terminating statement*>.

For an operation defined outside of an <*agent definition*> none of the <*statement*> items contained in its <*operation statements*> (either directly contained or indirectly, through <*statements*>) shall perform a queuing or blocking operation (sending signals, setting timers, etc.) or be a <*transition terminating statement*>.

Model

An <*operation definition*> with virtuality is a virtual operation and may be redefined. The <*operation definition*> in the redefinition replaces the <*operation definition*> in the supertype of the containing type.

If a <*list type*> is present in a <*parameter*>, this <*parameter*> is transformed as specified in clause I.4.1.

An <*operation definition*> that is a method is shorthand for an <*operation definition*> that is an operator. An <*in parameter*> with an anonymous <*name*>, the <*type*> of the containing <*class definition*> and <*aggregation*> `ref` is inserted as the first <*parameter*> into <*parameters*>.

The <*variable access*> `this` is transformed into a <*variable access*> referencing this anonymous <*parameter*>.

Mapping

An <*operation definition*> represents an Operation with isOperator false. The containing <*entity*> represents owner; the <*name*> represents name; each <*parameter*> in <*parameters*> of <*operation signature*> represents ownedParameter; the <*result*> represents ownedParameter. The <*type*> of <*result*> represents the type. If the <*operation definition*> defines a method, isLeaf is false. Otherwise, isLeaf is true.

If the <*operation definition*> redefines an <*operation definition*> in a supertype, the redefined <*operation definition*> represents redefinedOperation.

If the <*operation definition*> defines a procedure, it represents a StateMachine with exactly one Region with a single Transition. The source of the Transition is a Pseudostate with kind initial; the target is a FinalState; the <*operation statements*> represent the effect. This StateMachine represents method. If the <*operation definition*> defines a method or operator, the <*operation statements*> represent the node of an Activity. This Activity represents method.

The ownedParameter of the Operation are also used as the ownedParameter items of the Activity or StateMachine that serves as the method.

If an <*operation definition*> defining a procedure has a <*template*>, the <*context parameters*> represent the formalContextParameterList of the StateMachine that is the method of the represented Operation, as defined in clause I.4.4.2.

NOTE 4 – Operation definitions with a <*template*> are only supported for procedures.

Each <*name*> in the <*name*> list of a <*parameter*> represents Parameter with the isAnchored property being false. If present, <*aggregation*> represents aggregation, otherwise aggregation is composite; <*immutability*>, if present, represents the isReadOnly property being true, otherwise, the isReadOnly property is false; <*type*> represents type; the <*expression*> in <*default*>, if present, represents defaultValue. For an <*in parameter*>, direction is in; for an <*out parameter*>, direction is out; for an <*inout parameter*>, direction is inout.

A <*result*> represents Parameter with the isAnchored property being false. If present, <*aggregation*> represents aggregation, otherwise aggregation is composite; <*immutability*>, if present, represents the isReadOnly property being true, otherwise, the isReadOnly property is false; <*type*> represents type; direction is return.

I.9.1.6 Attribute

An attribute is a property of a type which specifies accessor and mutator operations for a data item associated with an instance of a type. The attribute may specify a data holder the data item will be associated with. The accessor and mutator operations will obtain and update the data item without making explicit to a caller how the data item is represented in the type.

An instance of a type with an attribute may store private data in a data holder. However, from the viewpoint of the client of the type, an attribute merely provides accessor and mutator operations, that is, an operation to access the data item associated with the data holder, and an operation to change the data item associated with the data holder. The attribute does not, however, provide operations that operate directly on the associated data item. Consequentially, the class may later change the manner in which it realizes its interfaces without the clients of the class having to change. As long as the class still provides the operations equivalent to the accessor and mutator operations to its clients, the class may change or remove an attribute, without affecting its clients.

NOTE 1 – If a class needs to enable its clients to manipulate the content of the data item associated with an attribute, appropriate operations should be provided by the class, which access and change the content of that data item.

NOTE 2 – Based on the operations of a class, a client of the class cannot discern whether and how private data of the class is stored in attributes.

Concrete grammar

An <*operation definition*> specifies an accessor for a <*type*>, if the <*type*> contains an <*attribute definition*> with <*name*> and this <*name*> is the <*name*> of the <*operation definition*>, the <*parameters*> of the <*attribute definition*>, if any, are the <*parameters*> of the <*operation definition*>, <*type*> is the <*result*> type of the operation and has the same <*modifier*> as in the <*attribute definition*>, and if a mutator is specified for this <*type*> with this <*name*>, the condition below is met.

An <*operation definition*> specifies a mutator for a <*type*>, if the <*type*> contains an <*attribute definition*> with <*name*> and this <*name*> is the <*name*> of the <*operation definition*>, the <*parameter*> items in the <*parameters*> of the <*attribute definition*>, if any, are the initial

<*parameter*> items of the <*operation definition*> and the last <*parameter*> in the <*parameters*> of the <*operation definition*> has <*type*> as its type, the <*result*> is omitted, and if an accessor is specified for this <*type*> with this <*name*>, the condition below is met.

When the mutator of a type is invoked on an instance of that type with <*expression*> as the last actual parameter, and the accessor for this type with this <*name*> is subsequently invoked on that instance, then the result of the accessor invocation shall be equal to the result of interpreting the <*expression*>.

<*attribute definition*> ::=
    [<*modifier*>] <*name*>+[| `,`] `:` <*type*> <*stereotype*>* [<*default*>] `;`
    | <*pseudo attribute*>

If a <*default*> is present, it shall be type compatible with <*type*>.

NOTE 3 – <*default*> is further explained in clause I.9.3.1.

If <*type*> identifies a syntype and a <*default*> is present, the result of the <*default*> shall be valid for the <*constraint*> of the syntype.

The redefinition of an attribute shall not change the <*type*> of the virtual attribute it redefines.

<*pseudo attribute*> ::=
    <*modifier*> <*name*> <*parameters*> `:` <*type*> <*stereotype*>* `;`

If an accessor operation is specified for a <*pseudo attribute*> within the same <*class definition*>, the <*modifier*> of its <*result*> shall be the <*modifier*> of the <*pseudo attribute*>.

Model

When a <*modifier*> is not present, the shorthand of clause I.9.3.1 applies. If visibility is absent, the stereotype <<`private`>> is inserted into the <*attribute definition*>.

For a <*pseudo attribute*>, the stereotype <<`abstract`>> is associated with the <*attribute definition*>, if not already present.

For each <*name*> of an <*attribute definition*> that is not a <*pseudo attribute*> and which is not abstract and not contained in an <*interface definition*>, a <*variable definition*> with the same <*name*>, <*type*> and <*default*>, if any, is constructed. The <*stereotype*> list applies to the <*variable definition*>, except for visibility stereotypes. The stereotype <<`private`>> is applied to the <*variable definition*>. If a <*default*> is present, it is used as the <*default*> for the <*variable definition*>. Otherwise, if no <*default*> is present, the <*variable definition*> does not have a <*default*>.

NOTE 4 – Consequentially, in a subtype of the class, the data holder could be realized in a different manner, as long as the accessor and mutator operations are provided giving the same interface as in the supertype.

NOTE 5 – A <*pseudo attribute*> does not create a <*variable definition*>. Therefore, the manner in which the data item is represented is determined by the implied accessor and mutator operations, which shall be provided in the containing class or a subtype.

For each <*name*> of an <*attribute definition*>, an accessor operation is constructed for the attribute <*type*> with this <*name*>, empty <*parameters*> and the <*modifier*> as result <*modifier*>, unless an <*operation definition*> with this <*name*> and operation signature is already present. If such <*operation definition*> is present without <*operation statements*> but where the result <*modifier*> differs from the <*modifier*> of the <*attribute definition*>, the result <*modifier*> is used as the result <*modifier*> of the implied accessor. The <*stereotype*> list in <*attribute definition*> applies to the accessor <*operation definition*>. If the <*attribute definition*> has the stereotype <<`abstract`>> associated and no accessor operation with operation statements is present, the accessor is abstract and does not have <*operation statements*>. Otherwise, the accessor operation returns the data item associated with the variable <*name*>.

For each <*name*> of an <*attribute definition*> that does not have **const** <*immutability*>, a mutator operation is constructed for the attribute <*type*> with this <*name*> and a single <*parameter*> with <*type*> as parameter <*type*> and <*aggregation*> **part**, unless an <*operation definition*> with this operation signature is already present. The <*stereotype*> list in <*attribute definition*> applies to the mutator <*operation definition*>. If the <*attribute definition*> has the stereotype <<abstract>> associated and no mutator operation is present, the mutator is abstract and does not have <*operation statements*>. Otherwise, the mutator operation assigns the <*parameter*> data item to the variable <*name*>.

After these transformations have been applied, the <*attribute definition*> is deleted.

NOTE 6 – The impact of the modifier, if present, on the variables created for each attribute definition is given in clause I.9.3.1.

An <*attribute definition*> with virtuality is a virtual attribute and may be redefined. The <*attribute definition*> in the redefinition replaces the <*attribute definition*> in the supertype of the containing type.

Mapping

For each <*name*> in the <*name*> list of an <*attribute definition*>, the <*attribute definition*> represents a Property with initialNumber being 0 and <*name*> representing name. If present, <*aggregation*> represents aggregation, otherwise aggregation is composite; <*type*> represents type; the <*expression*> in <*default*>, if present, represents defaultValue. If the <*attribute definition*> defines a read-only attribute, isReadOnly is true.

I.9.1.7 Syntype

A syntype specifies a subset of the elements of a type. A syntype used as a type has the same semantics as the parent type except for checks that data items belong to the specified subset of the elements of the parent type.

Concrete grammar

```
<syntype definition> ::=
        syntype <name> <stereotype>* = <parent type> ;

<parent type> ::=
        <type>
```

The <*parent type*> shall reference a <*class definition*>.

When the <*parent type*> is in turn defined by a <*syntype definition*>, the two syntypes shall not be mutually defined, that is, the <*parent type*> of a <*syntype definition*> shall not refer directly or indirectly to the syntype being defined.

If the <*parent type*> identifies a <*constrained type*> where the <*type identifier*> references a syntype, its <*constraint*> is not restricted to the range of the referenced syntype, if any.

The <*parent type*> in a redefined syntype shall be the same type or a subtype of the virtual syntype it redefines. If the range check specified by the <*constraint*> in the redefined syntype holds for a data item, then the range check specified by the <*constraint*> in the virtual syntype it redefines shall also hold for that data item.

Model

A <*syntype definition*> with virtuality is a virtual syntype and may be redefined. The <*syntype definition*> in the redefinition replaces the <*syntype definition*> in the supertype of the containing type.

Mapping

A <*syntype definition*> represents <<`Syntype`>><u>Class</u>. The <*name*> represents <u>name</u>. The <*parent type*> represents <u>supplier</u>. If the <*parent type*> has a <constraint>, it represents <u>constraint</u>.

I.9.1.8 Constraint

A constraint defines a range check, that is, an operation that determines whether a given data item satisfies the constraint.

Concrete grammar

<*constraint*> ::=
    `[` {<*range constraint*> | <*size constraint*>} **`]`**

<*range constraint*> ::=
    <*range*>+[| **`,`**]

If all <*expression*> items occurring in the <*range*> items are of the same type as the type being constrained (that is, the <*parent type*> of a <*syntype definition*> or the type of the <*expression*> in a <*decision statement*>), the <*constraint*> specifies a range constraint.

<*size constraint*> ::=
    <*range*>

If the <*constraint*> does not specify a range constraint and any <*expression*> occurring in the <*range*> is of the predefined type **`Natural`**, the <*constraint*> is a size constraint.

<*enumeration*> ::=
    `[` <*range constraint*> **`]`**

All <*expression*> items occurring in the <*range*> items of the <*range constraint*> shall be of the same type.

<*range*> ::=
    <*closed range*>
   | <*open range*>
   | <*expression*>
   | <*unconstrained range*>

<*unconstrained range*> ::=
    ★

<*closed range*> ::=
    <*lower bound*> **`..`** <*upper bound*>

<*lower bound*> ::=
    <*expression*>

<*upper bound*> ::=
    <*expression*> | ★

If a <*range*> is a <*closed range*> and the range check is a range constraint, the type the range check is applied to shall support the operation **`'<'`**.

<*open range*> ::=
    {**`==`** | **`!=`** | **`<`**| **`>`** | **`<=`** | **`>=`** } <*expression*>

If a <*range*> is an <*open range*> containing **`<`**, **`>`**, **`<=`** or **`>=`** and the range check is a range constraint, the type the range check is applied to shall support the operation **`'<'`**.

Model

If a <*range*> occurs in a <*constraint*>, it is transformed as follows: a <*range*> consisting of a single <*expression*> is transformed into an <*open range*> with the operation `'=='` and the <*expression*> as <*expression*>. A <*closed range*> where <*upper bound*> is `*` is transformed into an <*open range*> with the operation `'>='` and the <*expression*> in <*lower bound*> as <*expression*>. An <*unconstrained range*> is removed.

Mapping

A <*range constraint*> or <*size constraint*> represents a RangeCondition. Each <*range*> represents a ConditionItem in the conditionItemSet, such that the owner is the RangeCondition.

A <*closed range*> represents ClosedRange. The <*lower bound*> represents firstConstant. The <*upper bound*> or asterisk symbol represents secondConstant.

An <*open range*> represents an OpenRange. The symbol `==`, `!=`, `<`, `>`, `<=` or `>=` represents the operationIdentifier. The <*expression*> represents the constantExpression.

I.9.2 Use of data

This clause defines expressions and how literals and operations are interpreted in expressions.

NOTE – The use of expressions that depend on variables is defined in clause I.9.3. The use of expressions that depend on the state of agents is defined in clause I.9.4.

I.9.2.1 Expression

The interpretation of an expression returns a data item. The data item returned by the expression may depend on the state of the system or the interpretation of other elements of the specification or it may be independent of these (that is, it may be a constant).

Concrete grammar

<*expression*> ::=
    <*type coercion expression*>
  | <*conditional expression*>
  | <*disjunctive expression*>
  | <*expression*> **`xor`** <*expression*>
  | <*conjunctive expression*>
  | <*equality expression*>
  | <*expression*> {`<` | `>` | `<=` | `>=` | **`in`** } <*expression*>
  | <*type check expression*>
  | <*operand*>

<*operand*> ::=
    <*operand*> {`+` | `-` } <*operand*>
  | <*operand*> {`*` | `/` | **`mod`** | **`rem`** } <*operand*>
  | <*term*>

<*term*> ::=
    `-` <*term*>
  | <*negation expression*>
  | <*imperative expression*>
  | <*term*> {`++` | `--` }
  | <*primary*>

<*primary*> ::=
    <*operation call*>
  | <*target expression*>

<*target expression*> ::=
<*variable access*>
| <*literal expression*>
| <*agent expression*>
| <*create request*>
| <*compound expression*>
| <*parenthesized expression*>

<*compound expression*> ::=
`({` <*statement*>* <*expression*> `})`

<*parenthesized expression*> ::=
`(` <*expression*> `)`

The productions for <*expression*>, <*operand*> and <*term*> offer special syntactic forms for <*quoted name*> items. These syntactic forms are introduced so that common operations (for example, arithmetic and Boolean operations) have their usual syntactic appearance. Consequentially, one may write `(1 + 1) = 2` rather than being forced to use, for example, `equal(add(1,1),2)`.

A <*prefix operation name*>, <*infix operation name*> or <*postfix operation name*> in an expression, operand or term has the normal semantics of an operation but with prefix, infix or postfix syntax, respectively. The <*quoted name*> may also be used in standard operation notation.

The order of precedence of <*infix operation name*>, <*prefix operation name*> and <*postfix operation name*> items determines the binding of operations and is shown in Table I.9.1. An <*operation identifier*> with lower precedence binds tighter than an <*operation identifier*> with higher precedence. When the binding is ambiguous, then binding is from left to right.

Table I.9.1 – Precedence of special quoted operation names

Quoted operation name	Precedence
`':'`	11
`'?:'`	10
`'||'`, `'xor'`	9
`'&&'`	8
`'=='` , `'!='`	7
`'<'`, `'>'`, `'<='`, `'>='`, `'in'`, `':?'`	6
`'+'`, `'-'`	5
`'*'`, `'/'`, `'mod'`, `'rem'`	4
`'-'`, `'!'`	3
`'++'`, `'--'`	2
`'#'`	1

An <*expression*> is assignable if it is an <*identifier*> referencing a variable without <*immutability*> or with `val` <*immutability*>, or it is an <*operation call*> such that its <*name*> references a mutator operation (see clause I.9.1.5) and the parameters match all but the last parameter of the mutator, or it is an <*operation call*> that is a <*method call*> such that its rightmost invocation is a <*name*> referencing a mutator operation and the parameters match all but the last parameter of the mutator.

NOTE 1 – An expression that is an <*operation call*> of a mutator operation created by the transformation in clause I.9.1.6 is assignable.

An <*expression*> is able to be referenced if it is an <*identifier*> referencing a <*variable definition*> that does not contain <*immutability*>, or it is an <*operation call*> such that its <*name*> references an accessor operation (see clause I.9.1.5), or it is an <*operation call*> that is a <*method call*> such that its rightmost invocation is a <*name*> referencing an accessor operation, and the accessor operation does not have <*immutability*>.

An <*expression*> is a constant, if the data item returned by the <*expression*> does not depend on the current data items associated with variables that do not have <*immutability*> **`const`**, or on the existence or identities of agents or the system state (such as imperative expressions). The result of a constant expression is independent of when it is interpreted.

NOTE 2 – Some kinds of <*expression*> items (such as a <*literal expression*> or a <*variable access*> to a variable with <*immutability*> **`const`**) are always constant. Other kinds of expressions (such as any other <*variable access*>) always depend on the interpretation. Expressions that have other expressions as elements (such as an <*operation call*>) are constant if all their elements are constant.

Model

An <*expression*> of the form of an <*expression*> followed by an <*infix operation name*>, followed by an <*expression*> is shorthand notation for an <*operation call*> with the <*infix operation name*> converted into a <*quoted name*> as <*name*> and the <*expression*> items as <*actual parameter*> items.

An <*expression*> of the form of a <*prefix operation name*> followed by an <*expression*> is shorthand notation for an <*operation call*> with the <*prefix operation name*> converted into a <*quoted name*> as <*name*> and the <*expression*> as <*actual parameter*>.

An <*expression*> of the form of an <*expression*> followed by a <*postfix operation name*> is shorthand notation for an <*operation call*> with the <*postfix operation name*> converted into a <*quoted name*> as <*name*> and the <*expression*> as <*actual parameter*>.

An operation name in an infix, prefix or postfix <*expression*> is converted into a <*quoted name*> by surrounding it with apostrophe symbols.

A <*boolean conditional expression*> is shorthand notation for an <*operation call*> with **`'?:'`** as <*name*> and the <*expression*>, <*consequence expression*> and <*alternative expression*> items as <*actual parameter*> items.

For a <*compound expression*>, an <*operation definition*> with anonymous <*name*> is constructed within the namespace in which the <*compound expression*> occurs, such that its body consists of the <*statement*> list, augmented with a <*return statement*> with <*expression*> as the <*expression*> in its <*return*>. For every reference to a <*variable definition*> within the current namespace, an <*inout parameter*> is constructed, and this <*parameter*> is inserted into <*parameters*>, in the order of occurrence. The <*compound expression*> is then replaced by an <*operation call*> invoking this anonymous operation, with the references to <*variable definition*> items as <*actual parameters*> in the order of occurrence, if any.

Mapping

An <*expression*> represents a subtype of SdlExpression, as specified for each kind of <*expression*>. If the <*expression*> is a constant expression, isConstant is true; otherwise isConstant is false.

A <*parenthesized expression*> represents what its <*expression*> represents.

I.9.2.2 Literal expression

A literal is a specific instance of a type or the special symbol **`null`**.

Concrete grammar

<*literal expression*> ::=
    <*identifier*> | <*null*>

<*null*> ::=
    `null`

Whenever a <*literal expression*> is specified with an <*identifier*>, the unique <*name*> is derived in the same way, with the result type derived from context. A literal <*name*> is derived from context (see clause I.2.2) so that if the <*literal expression*> is overloaded (that is, the same <*name*> is used for more than one literal or operation), then the <*literal expression*> identifies a visible <*name*> with the result type consistent with the literal expression. If there are two literals with the same <*name*> but differing by result types, these are different.

It shall be possible to bind each <*literal expression*> containing an <*unqualified name*> to exactly one defined literal <*name*>.

Mapping

A <*literal expression*> represents a LiteralValue. Its value is represented by the referenced <*simple name*>, <*string name*>, <*real name*>, <*integer name*> or by <*null*>. The type of the <*literal expression*> represents type.

I.9.2.3 Equality and identity

An <*equality expression*> represents the equality of the data items returned by its expressions.

Concrete grammar

<*equality expression*> ::=
    <*positive equality expression*>
    | <*negative equality expression*>

<*positive equality expression*> ::=
    <*expression*> **`==`** <*expression*>

<*negative equality expression*> ::=
    <*expression*> **`!=`** <*expression*>

The type of one of the <*expression*> items of an <*equality expression*> shall be type compatible to the type of the other <*expression*>.

Mapping

An <*equality expression*> represents an EqualityExpression. The first <*expression*> represents firstOperand. The second <*expression*> represents secondOperand. If the <*equality expression*> is a <*positive equality expression*>, the operationIdentifier is **`=`**. If the <*equality expression*> is a <*negative equality expression*>, the operationIdentifier is **`!=`**.

I.9.2.4 Boolean expression

A Boolean expression is a Boolean disjunction or conjunction interpreted in short-circuit manner.

Concrete grammar

<*disjunctive expression*> ::=
    <*expression*> || <*expression*>

<*conjunctive expression*> ::=
    <*expression*> **`&&`** <*expression*>

For a <*disjunctive expression*> or <*conjunctive expression*>, both <*expression*> items shall be of the predefined `Boolean` type.

<*negation expression*> ::=
    `!` <*term*>

For a <*negation expression*>, the <*term*> shall be of the predefined `Boolean` type.

Model

A <*disjunctive expression*> is shorthand for a <*conditional expression*> where the <*expression*> is the first <*expression*>, the <*consequence expression*> is the predefined `Boolean` value `true` and the <*alternative expression*> is the second <*expression*>.

A <*conjunctive expression*> is shorthand for a <*conditional expression*> where the <*expression*> is the first <*expression*>, the <*consequence expression*> is the second <*expression*>, and the <*alternative expression*> is the predefined `Boolean` value `false`.

A <*negation expression*> is shorthand for a <*conditional expression*> where the <*expression*> is the <*term*>, the <*consequence expression*> is the predefined `Boolean` value `false` and the <*alternative expression*> is the predefined `Boolean` value `true`.

Mapping

NOTE – The interpretation of the <*conditional expression*> is given in the Model.

I.9.2.5 Conditional expression

A conditional expression is an expression where a Boolean expression is interpreted to determine whether to interpret a consequence or an alternative expression.

Concrete grammar

<*conditional expression*> ::=
    <*boolean conditional expression*> | <*null check expression*>

<*boolean conditional expression*> ::=
    <*expression*> `?` <*consequence expression*> `:` <*alternative expression*>

<*null check expression*> ::=
    <*expression*> `null ?` <*consequence expression*>

<*consequence expression*> ::=
    <*expression*>

<*alternative expression*> ::=
    <*expression*>

For a <*boolean conditional expression*>, the type of the <*consequence expression*> shall be compatible to the type of the <*alternative expression*>, or the type of the <*alternative expression*> shall be compatible to the type of the <*consequence expression*>, and the type of the <*expression*> shall be the predefined type `Boolean`.

For a <*null check expression*>, the type of the <*consequence expression*> shall be compatible to the type of the <*expression*>.

Model

A <*null check expression*> is shorthand for a <*conditional expression*> where the <*expression*> is a <*positive equality expression*> with the <*expression*> as the first <*expression*> and `null` as the second <*expression*>, the <*consequence expression*> is the <*consequence expression*> and the <*alternative expression*> is the <*expression*>.

Mapping

A <*boolean conditional expression*> represents a ConditionalExpression. The <*expression*> represents booleanExpression. The <*consequence expression*> represents consequenceExpression. The <*alternative expression*> represents alternativeExpression.

NOTE – The interpretation of the <*null check expression*> is given in the Model above.

I.9.2.6 Operation call

An operation call causes the interpretation of the named operation. The result returned by the interpretation of the operation is the result of the operation call.

Concrete grammar

<*operation call*> ::=
    <*operator call*>
    | <*method call*>
    | <*procedure call*>
    | <*constructor call*>

If an <*identifier*> can be interpreted as either the <*identifier*> of an <*operator call*>, <*procedure call*> or <*constructor call*> with omitted <*actual parameters*> or as a <*variable access*>, it is interpreted as an operator, procedure or constructor <*identifier*> unless it occurs within the <*operation statements*> of an accessor <*operation definition*> with this <*identifier*> and empty <*parameters*>, as introduced in clause I.9.1.6.

NOTE 1 – It may be required to use a qualifier to identify the unique <*operation definition*> referenced by the <*identifier*> of an <*operator call*>, <*procedure call*> or <*constructor call*>. In particular, for an <*operation definition*> defined in a type with a <*template*> the intended <*operation definition*> may not be able to be determined otherwise.

<*operator call*> ::=
    <*operation identifier*> [<*actual parameters*>]

<*operation identifier*> ::=
    <*identifier*> | <*template instantiation*>

The <*operation identifier*> of the <*operator call*> shall identify an <*operation definition*> that is an operator or method (see clause I.9.1.5).

<*method call*> ::=
    <*target*> **.** <*invocation*>

<*target*> ::=
    { <*target expression*> | <*operator call*> | <*procedure call*> | <*constructor call*> }
        { **.** <*invocation*> }*

<*invocation*> ::=
    <*operation identifier*> [<*actual parameters*>]

A <*procedure call*> in a <*target*> shall reference an <*operation definition*> with a <*result*>. The <*operation identifier*> of the <*invocation*> shall identify an <*operation definition*> that is a method (see clause I.9.1.5).

<*procedure call*> ::=
    <*operation identifier*> [<*actual parameters*>]

The <*operation identifier*> of the <*procedure call*> shall identify an <*operation definition*> or <*constructor definition*> that is a procedure (see clause I.9.1.5).

<*constructor call*> ::=
    { <*operation identifier*> | <*type identifier*> } [<*actual parameters*>]

The <*operation identifier*> of the <*constructor call*> shall reference a <*constructor definition*>. If a <*type identifier*> is used in a <*constructor call*>, it references the unnamed <*constructor definition*> with matching <*parameters*> contained in the type definition with this <*type identifier*>.

<*actual parameters*> ::=
    `(` [<*actual parameter*>+[| `,`] | <*named actual parameter*>+[| `,`]] `)`

The number of the <*actual parameters*> shall match the number of elements required in the context where <*actual parameters*> is used. The corresponding list of <*parameters*> determines the number of required elements.

<*actual parameter*> ::=
    <*expression*> | <*omitted parameter*>

<*named actual parameter*> ::=
    <*name*> `:=` <*expression*>

Each <*expression*> that is an <*actual parameter*> corresponding to an <*in parameter*>, if present, shall be type compatible with the <*type*> in the corresponding (by position) <*parameter*> of the <*parameters*> of the identified <*operation definition*>, unless the <*actual parameter*> is an <*omitted parameter*>. Each <*expression*> that is an <*actual parameter*> corresponding to an <*out parameter*> or an <*inout parameter*>, if present, shall have the same type as the corresponding (by position) <*parameter*> of the <*parameters*> of the identified <*operation definition*>, unless the <*actual parameter*> is an <*omitted parameter*>.

The <*expression*> in an <*actual parameter*> corresponding by position to an <*in parameter*> in the identified <*operation definition*> or <*constructor definition*> shall satisfy the syntactic constraints on assignment (see clause I.9.3.3), that is, if the parameter has **`ref`** aggregation and the actual parameter has **`part`** aggregation, then the actual parameter expression shall be able to be referenced (see clause I.9.2.1), and if the parameter has **`part`** aggregation and is not a direct or indirect subtype of **`Agent`**, the actual parameter expression shall not be **`null`**.

The <*expression*> in an <*actual parameter*> corresponding by position to an <*out parameter*> or an <*inout parameter*> in the identified <*operation definition*> or <*constructor definition*> shall be assignable (see clause I.9.2.1).

If all the <*expression*> items in <*actual parameters*> are constant expressions and the <*operation statements*> of the identified <*operation definition*> or <*constructor definition*> do not contain <*imperative expression*> items or <*variable access*> items referencing variables outside of the <*operation statements*> or <*agent expression*> items, the <*operation call*> is a constant expression as defined in clause I.9.2.1.

<*omitted parameter*> ::=

    _

An <*omitted parameter*> shall be used as <*actual parameter*> only if <*default*> is present in the corresponding <*parameter*>.

Model

If the <*operation identifier*> of an <*operator call*> references an <*operation definition*> that is a method, the <*operator call*> is transformed into a <*method call*> with the <*target*> **`this`**, where the <*invocation*> is formed from the <*operation identifier*> and <*actual parameters*>, if any, of the <*operator call*>.

If the <*operation identifier*> or the <*type identifier*> of a <*constructor call*> with omitted <*actual parameters*> references a <*constructor definition*> with an empty <*parameter*> list as <*parameters*>, the <*constructor call*> is transformed into a <*constructor call*> with an empty <*actual parameter*> list as <*actual parameters*>.

If the <*operation identifier*> of an <*invocation*> with omitted <*actual parameters*> references an accessor <*operation definition*> (see clause I.9.1.6) with an empty <*parameter*> list as <*parameters*>, the <*invocation*> is transformed into an <*invocation*> with an empty <*actual parameter*> list as <*actual parameters*>.

NOTE 2 – After this transformation and the transformations in clause I.9.3.2 have been applied, the <*actual parameters*> of an <*operation call*> will always be present, albeit they may be empty.

If <named actual para*meter*> items are present in the <*actual parameters*>, then the <*actual parameter*> items are reordered to match against the <*parameters*> of the identified <*operation definition*> or <*constructor definition*> based on the <*name*> in the <named actual para*meter*> which shall be identical to the <*name*> of the corresponding <*parameter*>. Then all <named actual para*meter*> items are replaced by their <*expression*> items. If <named actual para*meter*> items are present in the <*actual parameters*> and an actual parameter corresponding to a <*parameter*> is omitted, an <*omitted parameter*> is inserted in the corresponding position in the <*actual parameters*>.

If an <*omitted parameter*> is used as <*actual parameter*> in the <*actual parameters*>, the <*expression*> provided in the <*default*> of the corresponding <*parameter*> is used as <*actual parameter*>.

If the <*operation call*> has a **`part`** return aggregation or the <*operation call*> is a <*method call*> with an <*operator call*>, a <*compound statement*> is created with the <*statement*> containing the <*operation call*> as its <*statements*>, if such <*compound statement*> does not already exist.

If the last <*parameter*> in the <*parameters*> of the referenced <*operation definition*> or <*constructor definition*> contains a <*list type*>, a <*variable definition*> with an anonymous <*name*> and the anonymous type created by the <*syntype definition*> corresponding to the <*list type*> (see clause I.4.1) is created. The <*actual parameter*> corresponding (by position) to that <*parameter*> and all subsequent <*actual parameter*> items are assigned to this variable as a collection, that is, an <*expression*> constructing an instance of the predefined type **`String`** containing these <*actual parameter*> items is assigned to this variable. Then the anonymous <*name*> of this anonymous variable is used as the <*actual parameter*> of the <*operation call*> instead of these <*actual parameter*> items.

If the <*operation identifier*> is an <*identifier*> and does not reference an <*operation definition*>, but there is an <*operation definition*> with a template and the <*name*> in <*identifier*> such that <*actual context parameters*> can be derived given the <*actual parameters*> of the <*operation call*> and the context, this is shorthand for an <*operation identifier*> that is a <*template instantiation*> with the <*identifier*> as <*base identifier*> and the derived <*actual context parameters*>.

If the <*operation identifier*> is an <*identifier*> and does not reference a <*constructor definition*>, but there is a type definition with a template and the <*name*> in <*identifier*> such that <*actual context parameters*> can be derived given the <*actual parameters*> of the <*operation call*> and the context, this is shorthand for a <*type identifier*> that is a <*template instantiation*> with the <*identifier*> as <*base identifier*> and the derived <*actual context parameters*>.

A <*method call*> is converted into an <*operator call*> by inserting <*target*> or <*operator call*> as the first <*actual parameter*> into <*actual parameters*> of the first <*invocation*> of the <*invocation*> list and deleting that <*invocation*>, or creating <*actual parameters*> if absent. This transformation is applied until the <*invocation*> list is empty.

Mapping

An <*operator call*>, <*procedure call*> or <*constructor call*> represents an OperationApplication. The <*identifier*> or <*type identifier*> represents the operationIdentifier; <*actual parameters*> represents actualParameterList.

The interpretation of a <*method call*> is given in the Model above.

Each <*expression*> in an <*actual parameter*> in <*actual parameters*> represents an SdlExpression in the actualParameterList. An <*omitted parameter*> represents an empty SdlExpression in the actualParameterList.

I.9.2.7 Type checking and coercion

A type check expression determines whether an expression is within the range of data items given by a type (possibly with a constraint) or syntype.

A type coercion expression changes the dynamic type of an expression.

Concrete grammar

<*type coercion expression*> ::=
    <*expression*> **:** <*type*>

The type of the <*expression*> shall be type compatible with the type identified by <*type*>.

The static type of the <*expression*> shall be the type identified by <*type*> or a supertype of that type.

<*type check expression*> ::=
    <*expression*> **:?** <*type*>

If <*type*> is a <*constrained type*>, this represents the application of a range check as described in clause I.9.1.8.

NOTE – If the <*expression*> in a <*type check expression*> is <*null*>, the <*type check expression*> will always return the predefined **Boolean** value **true**.

Mapping

A <*type coercion expression*> represents TypeCoercion, such that <*expression*> represents expression and <*type*> represents sortReferenceIdentifier.

A <*type check expression*> where <*type*> is a <*constrained type*> or references a <*syntype definition*> represents a RangeCheckExpression such that <*expression*> represents expression. If <*type*> is a <*constrained type*>, the <*range constraint*> or <*size constraint*> in the <*constraint*> of the <*constrained type*> represents rangeCondition and its <*type*> represents parentSortIdentifier. If <*type*> references a <*syntype definition*>, <*type*> represents parentSortIdentifier.

Otherwise, a <*type check expression*> represents a TypeCheckExpression, where <*expression*> represents expression and <*type*> represents parentSortIdentifier.

I.9.2.8 Imperative expression

Imperative expressions obtain results from the underlying system state, such as when accessing the system clock or the status of timers.

Concrete grammar

<*imperative expression*> ::=
    <*now expression*>
    | <*timer active expression*>
    | <*agent active expression*>
    | <*state check expression*>

<*now expression*> ::=
    now

<*timer active expression*> ::=
    active <*type identifier*> [<*actual parameters*>]

The types of the <*actual parameters*> shall correspond by position to the types of the <*parameters*> in the <*stimulus definition item*> of a <*timer definition*> with a <*type identifier*> matching the <*type identifier*>.

<*agent active expression*> ::=
    active <*agent expression*>

<*state check expression*> ::=
    state <*identifier*>

If the <*identifier*> of a <*state check expression*> is a <*qualified name*>, the <*qualifier list*> shall reference a <*service definition*> of the <*agent definition*> that contains the <*state check expression*>. Otherwise, the <*identifier*> in a <*state check expression*> shall reference a state in the containing service.

Mapping

A <*now expression*> item represents a NowExpression.

A <*timer active expression*> represents a TimerActiveExpression. The <*type*> represents the timerIdentifier. Each <*actual parameter*> in <*actual parameters*> represents an SdlExpression in the expression list.

An <*agent active expression*> represents an ActiveAgentExpression where <*agent expression*> represents agentIdentifier and isThis is false.

A <*state check expression*> represents an EqualityExpression where firstOperand is a StateExpression and secondOperand is the <*qualified name*> represented as the corresponding element of the predefined **Charstring** type.

I.9.3 Data holders

This clause defines the use data holders, and how an expression involving a data holder is interpreted.

A data holder has a type and possibly an associated data item of that type. By assigning a new data item to the data holder, the data item associated with the data holder is changed. The data item associated with the data holder can be used in an expression by accessing the data holder.

Any expression containing a data holder depends on the current state of the system, because the data item obtained by interpreting the expression varies according to the data item last assigned to the data holder.

Data holders are variables and the parameters and result of operations, because variables are implicitly created for these.

I.9.3.1 Variable

A variable is a data holder and is established by a variable definition. The variable exists as long as the scope containing the variable definition exists. When that scope is exited, the variable ceases to exist.

A variable has a modifier which governs the manner in which the variable is associated with a data item.

NOTE 1 – The meaning of variables, accessing of variables (see clause I.9.3.2) and assigning to variables (see clause I.9.3.3) specified applies also to implicitly created variables for parameters and result of operations, loop variables and input variables for transitions.

Concrete grammar

<*variable definition*> ::=
    [<*modifier*>] <*name*>+[| **,**] **:** <*type*> <*stereotype*>* [<*default*>] **;**

When an entity may be syntactically both an <*attribute definition*> or a <*variable definition*> and both the entity definition has the stereotype <<`private`>> applied and the operation definitions specified by the transformation in clause I.9.1.6 are present, then the entity shall be considered as a <*variable definition*>. Otherwise, the entity shall be considered as an <*attribute definition*>.

NOTE 2 – Typically, the top-level <*entities*> of a <*class definition*>, as specified by the user, will contain <*attribute definition*>, and <*variable definition*> items are created as specified in the transformation in clause I.9.1.6.

A <*variable definition*> shall not have an associated visibility stereotype.

<*modifier*> ::=
    [<*immutability*>] [<*optionality*>] [<*aggregation*>]

<*immutability*> ::=
    `const` | **`val`**

<*optionality*> ::=
    `opt`

<*aggregation*> ::=
    `part` | **`ref`**

If the <*immutability*> is **`const`**, the <*name*> of the <*variable definition*> shall not occur as the <*location*> of an <*assignment*>, unless this <*assignment*> occurs in a <*constructor initializer*>.

NOTE 3 – If a <*variable definition*> with <*immutability*> **`const`** does not have a <*default*>, this variable needs to be initialized in a <*constructor initializer*> of the containing <*class definition*>; otherwise it cannot be given a value.

When the type of a variable is a primitive type, its aggregation shall be **`part`**.

When the <*type*> of a variable is a type defined by an <*agent definition*>, its aggregation shall be **`part`**, and <*optionality*> shall not be present.

NOTE 4 – If a modifier consistent with these constraints is not present, it is added by the transformation below.

When **`ref`** aggregation is used, the <*type*> shall reference a <*class definition*> or an <*interface definition*>.

<*default*> ::=
    `:=` <*expression*>

If the <*immutability*> contains **`const`**, the <*expression*> in the <*default*> shall be a constant.

Model

When <*aggregation*> is not present, this is shorthand for the <*aggregation*> **`part`**.

If the <*type*> of a variable is a type defined by an <*agent definition*>, the <*default*> **`null`** is added, if no <*default*> is present.

If <*default*> is not present but the <*type*> references a type definition with a <*default constructor definition*>, an <*operation call*> to the default constructor is used as the <*default*>.

Mapping

A <*variable definition*> represents a Variable. If present, <*aggregation*> represents aggregation; otherwise aggregation is composite. If present, <*immutability*> **`const`** represents the isReadOnly property being true; otherwise, the isReadOnly property is false. The <*type*> represents type and <*name*> represents name.

An <*aggregation*> **`part`** represents AggregationKind composite. An <*aggregation*> **`ref`** represents AggregationKind none.

The <*optionality*> is not explicitly represented.

I.9.3.2 Accessing variables

This clause presents the mechanisms to retrieve the data item associated with a variable.

Concrete grammar

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

The <*identifier*> shall reference a <*variable definition*>, <*parameter*>, <*input variable*> or an implicitly defined variable.

The <*variable access*> **`this`** shall occur only in an <*operation definition*> that defines a method (see clause I.9.1.5) or a constructor.

Model

NOTE – The <*variable access*> items **`this`** and **`input`** are transformed as discussed in clauses I.9.1.5 and I.7.3.2, respectively.

Mapping

A <*variable access*> represents a VariableAccess. If an <*identifier*> is present, it represents the variable.

I.9.3.3 Assignment to variables

An assignment creates an association between an identified variable and the data item that is the result of interpreting an expression.

Concrete grammar

<*assignment*> ::=
    <*location*> **`:=`** <*expression*> **`;`**

The type of the <*expression*> shall be type compatible with the type of the <*location*>.

If the <*location*> is an <*identifier*> identifying a <*variable definition*> without <*optionality*>, the <*expression*> shall not be <*null*>, unless the <*type*> of the <*variable definition*> is a direct or indirect subtype of the predefined type **`Agent`**.

NOTE – If the <*expression*> has <*optionality*> but the <*location*> does not, a check whether the expression is **`null`** may need to be added before the <*assignment*> if it cannot be determined otherwise that the <*expression*> will return a result.

If the <*location*> has **`ref`** <*aggregation*>, and the <*expression*> has **`part`** aggregation, then the <*expression*> shall be able to be referenced (see clause I.9.2.1).

If the <*location*> has **`ref`** aggregation, the <*expression*> shall not have <*immutability*>.

<*location*> ::=
    <*identifier*>
  | <*operation call*>

Model

If the <*location*> is an <*identifier*> and references a mutator <*operation definition*> (see clause I.9.1.6) with <*identifier*> and a single <*parameter*> such that the <*expression*> is type compatible with that <*parameter*>, the <*assignment*> is transformed to a <*method call*> with <*identifier*> and <*expression*> as <*actual parameters*> in <*invocation*> and **`this`** as <*target*>, unless the <*assignment*> occurs within the <*operation statements*> of the corresponding mutator <*operation definition*>.

If the <*location*> is an <*operator call*> with <*identifier*> and <*actual parameters*>, and references a mutator <*operation definition*> with <*identifier*> such that each <*actual parameter*> is type compatible with the corresponding <*parameter*> of the mutator and the <*expression*> is type compatible with the last <*parameter*> of the mutator, the <*assignment*> is transformed to a <*method call*> with <*identifier*> and the <*actual parameter*> items and <*expression*> as <*actual parameters*> in <*invocation*> and **`this`** as <*target*>.

If the <*location*> is a <*method call*> with <*target*>, an <*invocation*> with <*identifier*> and omitted <*actual parameters*>, and <*identifier*> references a mutator <*operation definition*> with <*identifier*> and a single <*parameter*> such that the <*expression*> is type compatible with that <*parameter*>, the <*assignment*> is transformed to a <*method call*> with <*identifier*> and <*expression*> as <*actual parameters*> in <*invocation*> and <*target*>.

If the <*location*> is a <*method call*> with <*target*>, an <*invocation*> with <*identifier*> and <*actual parameters*>, and <*identifier*> references a mutator <*operation definition*> with <*identifier*> such that each <*actual parameter*> is type compatible with the corresponding <*parameter*> of the mutator and the <*expression*> is type compatible with the last <*parameter*> of the mutator, the <*assignment*> is transformed to a <*method call*> with <*identifier*> and the <*actual parameter*> items and <*expression*> as <*actual parameters*> in <*invocation*> and <*target*>.

Mapping

If <*location*> is an <*identifier*> that references an <*attribute definition*>, an <*assignment*> represents an AddStructuralFeatureValueAction; the <*location*> represents structuralFeature, and the <*expression*> represents value.

If <*location*> references a <*variable definition*> (not derived from an <*attribute definition*>) or a <*loop variable definition*>, an <*assignment*> represents an AddVariableValueAction; the <*location*> represents variable, and the <*expression*> represents value.

I.9.4 Expressions for agents and agentsets

It must be possible to identify an agent instance in order to send signals to it, store a newly created agent instance in an agentset, etc. This clause discusses the mechanisms of creating and accessing agent instances.

I.9.4.1 Accessing agents

Agent instances are accessed through agent identities associated with special variables that reflect the usage of an agent instance. For example, the agent instance that sent the most recently consumed signal can be determined by the **`sender`** variable. The agent instances identified with an agentset can be accessed by their index in the agentset.

Concrete grammar

<*agent expression*> ::=
  `self`
  | **`parent`**
  | **`offspring`**
  | **`sender`**
  | <*agent access*>

<*agent access*> ::=
  `@` <*agent location*>

<*agent location*> ::=
    <*identifier*> [**(** <*expression*> **)**]

The <*identifier*> in <*agent location*> shall refer to an <*agentset definition*> of the <*agent definition*> that contains the <*agent access*>. If the referenced <*agentset definition*> is a <*dynamic agentset*>, the <*expression*> in the <*agent location*> shall be present and shall be type compatible with the <*index type*>.

NOTE – It is not possible to access an <*agentset definition*> defined in the containing <*agent definition*> from within a nested <*agent definition*>.

If an <*agent location*> occurs in a <*collection enumeration*> or if the <*identifier*> of the <*agent location*> refers to a <*singleton agentset*>, the <*expression*> shall not be present. Otherwise, if the <*identifier*> of the <*agent location*> refers to a <*dynamic agentset*> the <*expression*> shall be present.

Model

An <*agent access*> is shorthand for accessing an implicitly declared variable implied by the agentset: if the <*qualified name*> of the <*agent location*> in an <*agent access*> references an <*agentset definition*> that is a <*dynamic agentset*> and an <*expression*> is present in the <*agent location*>, the identity of the active agent instance associated with the variable corresponding to the result of the <*expression*> implied by the agentset (see clause I.5.3) is returned. If the <*qualified name*> of the <*agent location*> in an <*agent access*> references an <*agentset definition*> that is a <*singleton agentset*>, the identity of the active agent instance associated with the variable implied by the agentset (see clause I.5.3) is returned.

If an active agent is not found at the <*agent location*>, the <*agent access*> returns **`null`** and does not identify any agent.

When an <*agent access*> appears in a <*collection enumeration*> (see clause I.8.4), the identities of the active agents in the agentset are returned as the items of the collection.

Mapping

An <*agent expression*> represents PidExpression. The keywords **`self`**, **`parent`**, **`sender`** or **`offspring`** represent expressionKind.

I.9.4.2 Create

A <*create request*> is used to create instances of agents (active data items). The identity of a created agent instance is always associated with an agentset as long as the agent is active.

NOTE 1 – Instances of classes (passive data items) are created by calling the constructor of the class.

Concrete grammar

<*create request*> ::=
    `create` <*operation call*> **@** <*agent location*>

The <*identifier*> in <*agent location*> shall refer to an <*agentset definition*> of the <*agent definition*> that contains the <*agent access*>. The <*identifier*> in the <*agent location*> of a <*create request*> shall not reference a <*singleton agentset*>.

NOTE 2 – It is not possible to create an agent instance in an <*agentset definition*> defined in the containing <*agent definition*> from within a nested <*agent definition*>.

The <*operation call*> shall be a <*constructor call*>.

Model

For a <*create request*>, a <*variable definition*> of an anonymous variable with type **`Agent`** is created and inserted before the statement in which the <*create request*> occurs.

For a <*create request*> with an <*expression*> in <*agent location*> and referencing a <*dynamic agentset*>, an <*if statement*> with an <*expression*> that is an <*agent active expression*> with the <*agent location*> of the <*create request*> as <*agent expression*> is inserted just before the statement in which the <*create request*> occurs. The <*statements*> of this <*if statement*> consist of a single <*assignment*> of `null` to the anonymous agent variable. The <*else branch*> consists of the <*create request*> as an <*expression statement*>; followed by, if an offspring was successfully created, an assignment of `offspring` to the implicitly declared variable corresponding to the result of the <*expression*> implied by the agentset (see clause I.5.3); followed by an <*assignment*> of `offspring` to the anonymous agent variable.

Otherwise, the <*create request*> is inserted as an <*expression statement*> just before the statement in which the <*create request*> occurs; followed by, if an offspring was successfully created, an assignment of `offspring` to an arbitrary implicitly declared variable implied by the agentset (see clause I.5.3) not associated with an active agent (for a <*dynamic agentset*>), or to the implicitly declared variable implied by the agentset (see clause I.5.3) (for a <*singleton agentset*>); followed by an <*assignment*> of `offspring` to the anonymous agent variable.

The original <*create request*> is replaced by the anonymous agent variable.

NOTE 3 – If an attempt is made to create an agent instance in an agentset at a location which identifies an active agent instance, then no new instance is created.

Mapping

A <*create request*> represents a CreateObjectAction where the operation identifier in the <*operator call*> represents classifier and each <*actual parameter*> in <*actual parameters*> represents an SdlExpression in actualParameterList.

I.10 Exceptions

An exception indicates that an exceptional situation (typically an error situation) has occurred while interpreting the specification. An exception instance is created implicitly by the underlying system.

Creation of an exception instance diverts the normal flow of control within the state machine of an agent instance or within an operation. If an exception instance is created and is not caught in the context where it was created, the exception instance propagates (dynamically) outwards to the calling or invoking context and is treated as if it were created at that context. If an exception instance propagates outwards until it reaches an agent instance and is not caught there, the further behaviour of the system is undefined.

A number of exceptions are predefined within the package `Predefined`. These exceptions may be created by the underlying system implicitly. A specification may also create instances of predefined exceptions explicitly.

I.11 Generic system definition

A system specification may have conditional parts that are selected depending on some externally defined condition or contain system parameters with unspecified results. Such a system specification is called generic. A generic system specification is tailored by selecting a suitable subset of the specification and providing a data item for each of the system parameters. The resulting system specification does not contain external conditions.

A generic system definition is a system definition that contains an externally defined condition (see clause I.11.1), or an operation defined by an external operation definition (see clause I.11.2) or external data. A system definition is created from a generic system definition by providing results for the external condition and providing behaviour for external operation definitions. How this is accomplished is not part of this appendix.

I.11.1 Conditionalization

Conditionalization allows the final contents of a specification to be determined based on external conditions, such as whether or not a name is defined. How the external name is defined and associated with the specification is not defined as part of this appendix.

Concrete grammar

<*selected entities*> ::=
  {**`#ifdef`** | **`#ifndef`** } <*selection*> <*selected entities item*>
  [**`#else`** <*selected entities item*>]
  `#endif`
  | **`#if`** <*selection condition*> <*selected entities item*>
  [**`#else`** <*selected entities item*>]
  `#endif`

<*selected entities item*> ::=
  <*stereotype*>* <*entity*>*

<*selected statements*> ::=
  {**`#ifdef`** | **`#ifndef`** } <*selection*> <*selected statements item*>
  [**`#else`** <*selected statements item*>]
  `#endif`
  | **`#if`** <*selection condition*> <*selected statements item*>
  [**`#else`** <*selected statements item*>]
  `#endif`

<*selected statements item*> ::=
  <*stereotype*>* { <*statement*> <*stereotype*>* }*

A <*selected entities item*> or <*selected statements item*> that is not selected need not be type correct nor need its <*identifier*> or <*type identifier*> items resolved to defined entities. The not selected items shall be otherwise syntactically correct.

A <*selected entities*> item may appear in any <*entities*> item of the specification. A <*selected statements*> item may appear in any <*statements*> in the specification. The <*selected entities*> and <*selected statements*> shall contain only those <*entity*> items and <*statement*> items that are syntactically allowed where they occur.

<*selection*> ::=
  <*name*>

<*selection condition*> ::=
  <*selection condition*> | | <*selection condition*>
  | <*selection condition*> **`&&`** <*selection condition*>
  | **`!`** <*selection condition*>
  | <*name*>
  | **`defined`** { **`(`** <*name*> **`)`** | <*name*> }
  | **`(`** <*selection condition*> **`)`**

To establish the order of precedence of the symbols in a <*selection condition*>, the symbols `||`, **`&&`** and **`!`** are treated as the operation name `'||'`, `'&&'` and `'!'`, respectively (see Table I.9.1).

Model

A <*selection*> with **`#ifdef`** is enabled if the <*name*> is externally defined. A <*selection*> with **`#ifndef`** is enabled if the <*name*> is externally not defined. A <*selection condition*> with **`defined`** is enabled if the <*name*> is externally defined. A <*selection condition*> that is a <*name*> is enabled if the <*name*> is externally defined and has a value different than `0`. A <*selection condition*> containing `!`, `&&` or `||` is interpreted as discussed in clause I.9.2.4, with an enabled <*selection condition*> being treated as a predefined `Boolean` value `true`, and a <*selection condition*> which is not enabled being treated as a predefined `Boolean` value `false`.

For the <*selected entities*>, if the <*selection*> or <*selection condition*> is enabled, the <*selected entities*> are replaced with the first <*selected entities item*>; otherwise, the <*selected entities*> are replaced with the second <*selected entities item*>, if present.

For the <*selected statements*>, if the <*selection*> or <*selection condition*> is enabled, the <*selected statements*> are replaced with the first <*selected statements item*>; otherwise, the <*selected statements*> are replaced with the second <*selected statements item*>, if present.

Mapping

The result of the Model for <*selected entities*> is mapped as the resulting <*entity*> (see clause I.3.1). The result of the Model for <*selected statements*> is mapped as the resulting <*statement*> (see clause I.8.1).

I.11.2 External definition

An external entity is defined by a definition whose body is not included in the specification. An external definition is treated as if part of the specification. How an external entity is interpreted is not further defined in this appendix.

Concrete grammar

The stereotype <<`extern`>> references a definition external to the specification.

When the stereotype <<`extern`>> is used in a definition, the body of the definition shall be empty.

Bibliography

[b-ITU-T Z.111]	Recommendation ITU-T Z.111 (2008), *Notations and guidelines for the definition of ITU-T languages*.
[b-ISO/IEC 10646]	ISO/IEC 10646:2012, *Information technology – Universal Coded Character Set (UCS)*.

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	Terminals and subjective and objective assessment methods
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, 2013