International Telecommunication Union

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Z.100
Annex F2
(10/2016)

SERIES Z: LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

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

Specification and Description Language – Overview of SDL-2010

Annex F2: SDL-2010 formal definition: Static semantics

Recommendation ITU-T Z.100 – Annex F2

International Telecommunication Union

ITU-T Z-SERIES RECOMMENDATIONS

LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

FORMAL DESCRIPTION TECHNIQUES (FDT)	
Specification and Description Language (SDL)	**Z.100–Z.109**
Application of formal description techniques	Z.110–Z.119
Message Sequence Chart (MSC)	Z.120–Z.129
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.100

Specification and Description Language – Overview of SDL-2010

Annex F2

SDL-2010 formal definition: Static semantics

Summary

Annex F2 describes the static semantic constraints of SDL-2010, and it also describes the transformations identified by the 'Model' clauses of Recommendations ITU-T Z.101, Z.102, Z.103, Z.104, Z.105 and Z.107, that are included by reference in Recommendation ITU-T Z.100.

History

Edition	Recommendation	Approval	Study Group	Unique ID*
1.0	ITU-T Z.100	1984-10-19		11.1002/1000/2222
1.1	ITU-T Z.100 Annex A	1984-10-19		11.1002/1000/6664
1.2	ITU-T Z.100 Annex B	1984-10-19		11.1002/1000/6665
1.3	ITU-T Z.100 Annex C1	1984-10-19		11.1002/1000/6666
1.4	ITU-T Z.100 Annex C2	1984-10-19		11.1002/1000/6667
1.5	ITU-T Z.100 Annex D	1984-10-19		11.1002/1000/6668
2.0	ITU-T Z.100	1987-09-30	X	11.1002/1000/10954
2.1	ITU-T Z.100 Annex A	1988-11-25		11.1002/1000/6669
2.2	ITU-T Z.100 Annex B	1988-11-25		11.1002/1000/6670
2.3	ITU-T Z.100 Annex C1	1988-11-25		11.1002/1000/6671
2.4	ITU-T Z.100 Annex C2	1988-11-25		11.1002/1000/6672
2.5	ITU-T Z.100 Annex D	1988-11-25	X	11.1002/1000/3646
2.6	ITU-T Z.100 Annex E	1988-11-25		11.1002/1000/6673
2.7	ITU-T Z.100 Annex F1	1988-11-25	X	11.1002/1000/3647
2.8	ITU-T Z.100 Annex F2	1988-11-25	X	11.1002/1000/3648
2.9	ITU-T Z.100 Annex F3	1988-11-25	X	11.1002/1000/3649
3.0	ITU-T Z.100	1988-11-25		11.1002/1000/3153
3.1	ITU-T Z.100 Annex C	1993-03-12	X	11.1002/1000/3155
3.2	ITU-T Z.100 Annex D	1993-03-12	X	11.1002/1000/3156
3.3	ITU-T Z.100 Annex F1	1993-03-12	X	11.1002/1000/3157

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

3.4	ITU-T Z.100 Annex F2	1993-03-12	X	11.1002/1000/3158
3.5	ITU-T Z.100 Annex F3	1993-03-12	X	11.1002/1000/3159
3.6	ITU-T Z.100 App. I	1993-03-12	X	11.1002/1000/3160
3.7	ITU-T Z.100 App. II	1993-03-12	X	11.1002/1000/3161
4.0	ITU-T Z.100	1993-03-12	X	11.1002/1000/3154
4.1	ITU-T Z.100 (1993) Add. 1	1996-10-18	10	11.1002/1000/3917
5.0	ITU-T Z.100	1999-11-19	10	11.1002/1000/4764
5.1	ITU-T Z.100 (1999) Cor. 1	2001-10-29	17	11.1002/1000/5567
6.0	ITU-T Z.100	2002-08-06	17	11.1002/1000/6029
6.1	ITU-T Z.100 (2002) Amd. 1	2003-10-29	17	11.1002/1000/7091
6.2	ITU-T Z.100 (2002) Cor. 1	2004-08-29	17	11.1002/1000/356
7.0	ITU-T Z.100	2007-11-13	17	11.1002/1000/9262
8.0	ITU-T Z.100	2011-12-22	17	11.1002/1000/11387
8.1	ITU-T Z.100 Annex F1	2000-11-24	10	11.1002/1000/5239
8.2	ITU-T Z.100 Annex F2	2000-11-24	10	11.1002/1000/5576
8.3	ITU-T Z.100 Annex F3	2000-11-24	10	11.1002/1000/5577
8.4	ITU-T Z.100 Annex F1	2015-01-13	17	11.1002/1000/12354
8.5	ITU-T Z.100 Annex F2	2015-01-13	17	11.1002/1000/12355
8.6	ITU-T Z.100 Annex F3	2015-01-13	17	11.1002/1000/12356
9.0	ITU-T Z.100	2016-04-29	17	11.1002/1000/12846
9.1	ITU-T Z.100 Annex F1	2016-10-29	17	11.1002/1000/13040
9.2	ITU-T Z.100 Annex F2	2016-10-29	17	11.1002/1000/13041
9.3	ITU-T Z.100 Annex F3	2016-10-29	17	11.1002/1000/13042

Keywords

Specification and Description Language, SDL-2010, formal definition, Static semantics, shorthand transformations.

FOREWORD

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

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

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

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

NOTE

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

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

INTELLECTUAL PROPERTY RIGHTS

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

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

Table of Contents

Recommendation ITU-T Z.100

Specification and Description language – Overview of SDL-2010

Annex F2

SDL-2010 formal definition: Static semantics

F2.1 General information for the static semantics

An overview of the static semantics is described in clauses F1.2.1, F1.2.2 and F1.2.3 of Annex F1.

F2.1.1 Definitions used from Annex F1

The following definitions for the syntax and semantics of abstract state machines (ASM) are used within this annex (Annex F2). They are defined in Annex F1. They are introduced here for cross-referencing reasons together with the keyword **provided** that is used in transformation rules (see clause F2.2.1.4 Transformation rules).

> The keywords **case**, **choose**, **controlled**, **else**, **elseif**, **endcase**, **endif**, **endlet**, **if, let**, **monitored**, **of**, **otherwise**, **provided**, **then**.
>
> The domains *X*, *BOOLEAN*, *NAT*, *TOKEN*, *DefinitionAS1* and *DefinitionAS0*.
>
> The functions *take*, *undefined*, *true*, *false*, *empty*, *head*, *tail*, *last*, *length*, *toSet*, *parentAS1*, *parentAS0*, *parentAS0ofKind*, *parentAS1ofKind*, *isAncestorAS0*, *isAncestorAS1* and *replaceInSyntaxTree*.
>
> The operation symbols *, +, **-set**, **-seq**, =, ≠, ∧, ∨, ⇒, ⇔, ¬, ∃, ∀, >, ≥, <, ≤, +, -, *, /, **in**, ×, ⌒, ∪, ∩, \, ∈, ∉, ⊆, ⊂, | |, **U**, ∅, **mk-**, **s-**, **s2-**.

For more information about the syntax of ASM see Annex F1.

F2.2 Static semantics

In this annex, the static semantics of SDL-2010 is formalized. This entails defining static well-formedness rules for the concrete and abstract syntax, transformations of the concrete syntax and mappings from concrete- to abstract syntax. The concrete syntax is expressed using Abstract Syntax Notation level 0 (AS0), an abstraction of the SDL text-only phrase representation (SDL-PR) grammar (see [ITU-T Z.106]). The well-formedness rules are expressed in terms of first order predicate calculus. The transformations are represented as rewrite rules, comprising patterns, conditions (expressed using first order predicate calculus) and substitutions. The mappings are expressed using the abstract state machine formalism defined in Annex F1.

For the static semantics, the presentation of the grammar as described in Annex F1 is again used.

Context conditions (well-formedness rules) are reflected in the abstract syntax tree as relations from nodes to nodes. First order predicate calculus is used to express the relations as follows: The nodes of the AST are the objects of reasoning. Some functions are defined to retrieve nodes, e.g., the function *n.parentAS1* returns the parent node of *n*. This is explained in detail in Annex F1. Syntax structures are described on sets of nodes by quantifying over them. Predicates are defined over nodes showing the context-dependent rules of these nodes.

For example, in order to define that all the entities of the same type in a scope unit obey the rules of no duplicate appearance, the following static semantic rule is defined:

$$\forall d, d' \in \mathit{ENTITYDEFINITION1}: d.\mathit{entityKind1} = d'.\mathit{entityKind1} \wedge d \neq d' \Rightarrow d.\mathit{identifier1} \neq d'.\mathit{identifier1}$$

where:

d and *d'* represent any two abstract syntax tree nodes belonging to the set *ENTITYDEFINITION1*.

d.entityKind1 and *d'.entityKind1* get the entity kinds of the corresponding syntax constructs.

d.identifier1 and *d'.identifier1* get the full identifiers of the corresponding syntax constructs.

F2.2.1 General definitions

F2.2.1.1 Division of text

The static semantics is presented with the following division of text. Please find below the headings used and for each of the headings a short description of the contents.

Abstract syntax

This part is used to describe the abstract grammar as already defined within Recommendation ITU-T Z.100. There will be usually no comments in this section as it is copied as is from the language definition.

Conditions on abstract syntax

This part reflects the conditions that can be formulated on the abstract syntax level. The conditions are usually commented by the corresponding part of the language definition.

Concrete syntax

This part shows the concrete syntax. In fact, an abstraction of the concrete syntax, namely the AS0 as defined below, is used. There will be usually no comments in this section as it is copied from the language definition.

Conditions on concrete syntax

This part reflects the conditions that must be true for the concrete syntax (AS0 here). The conditions are usually commented by the corresponding part of the language definition.

Transformations

This part shows the transformations within the AS0. Please see below for the format of the rules. The transformations are usually commented by the corresponding part of the language definition.

Mapping to abstract syntax

This part shows how the transformed AS0 is mapped to AS1. If the mapping is straightforward, no comments are given.

Auxiliary functions

This part introduces auxiliary functions that are used later on to define the conditions on AS0 and the transformations. The aim and the definition of the functions are explained.

F2.2.1.2 Abstraction of the concrete grammar (AS0)

For the sake of the definition of the static semantics rules, a special format of the concrete grammar is used. This special format is called abstract syntax level 0 (AS0). It is an abstraction of the concrete grammar of the text-only phrase representation (SDL-PR), where all the unnecessary grammar items such as separators and terminal keywords are omitted. Moreover, the AS0 is slightly changed in order to be able also to represent an abstraction of the concrete graphical grammar.

The idea is that the AS0 is generated by a very simple parsing algorithm from the concrete grammars of SDL-PR.

The AS0 does not only represent the original syntactical structure, but it also forms a tree. To achieve this, the syntax constructors "::=" of Recommendation ITU-T Z.100 are replaced by an alias construct ("=") and a tree node constructor ("::"). Both these constructs are already defined in Z.100 in the scope of the abstract syntax, which is called AS1 here.

The metalanguage for the concrete grammar is defined in [ITU-T Z.111]. However, the notation used in this document to represent semantic subcategories differs from the definition in [ITU-T Z.111]. For example, instead of writing

<data type name>

the following sections will use

<data type<name>

This convention facilitates automatic extraction of the grammar from this document.

F2.2.1.3 Static conditions

Usually, the AS0 conditions are checked before the transformations start. However, some conditions are only valid after some transformation steps. This is indicated by preceding the corresponding condition with a numbering sign (e.g., "=4=>"), where the number in the arrow indicates the next transformation step. This means, a condition with the prefix "=4=>" is checked between the transformation steps 3 and 4. By default, conditions are preceded with "=1=>", i.e., they are checked before any transformations.

F2.2.1.4 Transformation rules

Transformations are represented by rewrite rules. Please find below the syntax for rewrite rules.

<rewrite rule> ::=
<pattern> [<provisions>] "=" <integer> "=>" <expression> { **and** <dependent transformation> }*

<dependent transformation> ::=
<expression> { "=>" | ":=" } <expression>

<provision> ::=
provided <Boolean<expression>

The pattern as well as the expression refer to the syntax as defined for ASM in Part 1 of Annex F. The non-terminal constructor names must all match a non-terminal in the concrete syntax. A variable is not allowed to appear more than once on either side. Variable names that appear on the right hand side must also appear on the left hand side. Furthermore, the pattern and expression patterns must be correctly typed and be of the same type.

A rule Pattern =i=> Expression is equivalent to an ASM rule of the form

```
choose v:DefinitionAS0
    case v
        Pattern': e:=CreateExpr(Expression)
        ReplaceIn(v.Parent, v, e)
```

In the definition above, *CreateExpr* means for every constructor of Expression an extend of the corresponding domain and the setting of the contents function to a corresponding **mk-** for the following sub pattern. The placeholder *ReplaceIn* means to replace v by e in the parent node of v. This does not cause problems as the syntax tree is a tree and it is always possible to find the parent and to replace one of its children.

Dependent transformation rules have a similar semantics. They are interpreted together with their main rule.

The integer in a rewrite rule means the transformation step this rule belongs to. The steps are described in clause F2.3.

The optional <provisions> of <rewrite rule> are Boolean expressions the <pattern> needs to satisfy for the rule to be applied.

We use one auxiliary function *newName* to construct new names during the transformation.

monitored *newName*: <name> → <name>

The constraint on this function is that it always returns a new unique name. However, the result is the same when the argument is the same unless the argument is *undefined*. For an *undefined* parameter a new unique name that is not already used within the syntax tree is provided.

NOTE – In the 2016 version of this Annex, *newname* is never used with an argument, which has the same meaning as *newname*(*undefined*) and therefore always produces a new unique name.

F2.2.1.5 Mapping rules

The mapping rules introduce a function.

Mapping: *DefinitionAS0* → *DefinitionAS1*

The definition of the function *Mapping* is formed by the concatenation of all the cases contained in all **Mapping** sections. This is preceded with the following header part and followed by an **endcase**.

Mapping(*a*: *DefinitionAS0*): *DefinitionAS1* =$_{\text{def}}$
case *a* **of**

This way the mapping function is defined step by step in the appropriate places in the **Mapping** sections. Each alternative of the mapping will thus be preceded by a bar ("|"), because it is one alternative of the *Mapping* function description.

F2.2.1.6 Predefinition

The following domains and functions are used throughout the static conditions for AS1 and concrete syntax.

F2.2.1.6.1 General functions

The function *bigSeq* is used to concatenate a sequence of sequences into one sequence.

bigSeq(*s*: *X*-**seq-seq**): *X*-**seq** =$_{\text{def}}$
if *s* = *empty* **then** *empty* **else** *s*. *head* ⌒ *bigSeq*(*s.tail*) **endif**

F2.2.1.6.2 Domain definitions for AS1

SCOPEUNIT1: the union of all the scope units in AS1.

SCOPEUNIT1=$_{\text{def}}$ *Package-definition*
- ∪ *Agent-definition*
- ∪ *Agent-type-definition*
- ∪ *Procedure-definition*
- ∪ *Signal-definition*
- ∪ *Composite-state-type-definition*
- ∪ *Data-type-definition*
- ∪ *State-node*

ENTITYDEFINITION1: the union of all the entity definitions in AS1.

ENTITYDEFINITION1=$_{\text{def}}$ *Package-definition*
- ∪ *Agent-definition*
- ∪ *Agent-type-definition*
- ∪ *Procedure-definition*
- ∪ *Composite-state-type-definition*
- ∪ *Channel-definition*

∪ *Gate-definition*
∪ *Signal-definition*
∪ *Timer-definition*
∪ *Variable-definition*
∪ *Data-type-definition*
∪ *State-node*
∪ *Syntype-definition*
∪ *Literal-signature*
∪ *Operation-signature*

ENTITYKIND1: the set of all the entity kinds in AS1.

ENTITYKIND1 $=_{def}$ {**agent**, **agent type**, **package**, **state**, **state type**, **procedure**, **variable**, **signal**, **timer**, **channel**, **gate**, **sort**, **exception**, **literal**, **operation**}

AGENTKIND1: the set of agent kinds in AS1.

AGENTKIND1 $=_{def}$ {**system**, **block**, **process**}

SIGNAL1 $=_{def}$ {*id*∈*Identifier*: *id.idKind1*∈{**signal**, **timer**}}

TYPEDEFINITION1 $=_{def}$ *Agent-type-definition*∪*Procedure-definition*∪*Composite-state-type-definition*∪ *Data-type-definition*

VALUE1 $=_{def}$ *Literal-signature*

F2.2.1.6.3 Domain definitions for AS0

ENTITYKIND0 $=_{def}$ {**package**, **agent**, **system**, **block**, **process**, **agent type**, **system type**, **block type**, **process type**, **channel**, **gate**, **signal**, **signal list**, **timer**, **sort**, **interface**, **type**, **procedure**, **remote procedure**, **variable**, **synonym**, **literal**, **operator**, **method**, **remote variable**, **state**, **state type**, **exception**}

TYPEDEFINITION0 $=_{def}$ <agent type definition> ∪ <composite state type definition> ∪
<data type definition> ∪ <procedure definition> ∪
<interface definition> ∪ <signal definition>

FORMALCONTEXTPARAMETER0 $=_{def}$ <agent type context parameter> ∪
<agent context parameter> ∪ <procedure context parameter> ∪
<remote procedure context parameter> ∪
<signal context parameter name> ∪ <variable context parameter names> ∪
<remotevariable contextparameter names> ∪ <timer context parameter name> ∪
<synonym context parameter name> ∪ <sort context parameter> ∪
<compositestate type context parameter> ∪
<gate context parameter> ∪ <interface context parameter name>

In order to deal with the predefined data, the following four domains are introduced.

PREDEFINEDDEFINITION0 $=_{def}$ *PREDEFINEDLITERAL0*∪*PREDEFINEDOPERATION0*∪*PREDEFINEDSORT0*

PREDEFINEDLITERAL0 $=_{def}$ represents all the literals defined within the package Predefined.

PREDEFINEDOPERATION0 $=_{def}$ represents all the operations defined within the package Predefined.

PREDEFINEDSORT0 $=_{def}$ represents all the sorts defined within the package Predefined.

ENTITYDEFINITION0 $=_{def}$ <package definition> ∪ <agent definition> ∪ <composite state> ∪
<textual typebased agent definition> ∪<synonym definition> ∪<channel definition>
<textual typebased state partition def> ∪ <syntype definition> ∪
<timer definition item> ∪ <signal list definition> ∪ <parameters of sort> ∪ <variables of sort> ∪
<literal signature> ∪ <operation signature> ∪ <textual gate definition> ∪

<remote variables of sort> ∪ *TypeDefinition0* ∪ *FormalContextParameter0*∪
PredefinedDefinition0

TypeReference0 =def <agent type reference> ∪ <composite state type reference> ∪
<procedure reference>

Reference0 =def <package reference> ∪ <block reference> ∪ <process reference> ∪
<composite state reference> ∪ <operation reference> ∪ *TypeReference0*

ScopeUnit0 =def <package definition> ∪ <agent definition> ∪ <operation definition> ∪
<composite state> ∪ <sort context parameter> ∪ <signal context parameter name> ∪
<compound statement> ∪*TypeDefinition0*

Signal0=def {*id*∈<identifier>: *id.idKind0*∈{**signal**, **timer**, **remote procedure**, **remote variable**}}

Symbol0=def {"<",">","<=",">=","Length"}

Context0=def <assignment> ∪ <decision> ∪ <expression>

Binding0 =def <name> × *EntityDefinition0*

BindingList0 =def *Binding0**

F2.2.1.6.4 Function definitions on AS1

The function *identifier1* is used to get the identifier with full qualifier for an entity definition in AS1.

identifier1(*d*: *EntityDefinition1*):*Identifier* =def
mk-*Identifier*(*d. fullQualifier1*, *d.entityName1*)

The function *fullQualifier1* is used to get the full qualifier for an entity definition in AS1.

fullQualifier1(*d*: *EntityDefinition1*): *Qualifier* =def
let *su= parentAS1ofKind*(*d*, *ScopeUnit1*) **in**
if *su = undefined* **then** *empty*
elseif *d.entityKind1*∈{**operation**, **literal**}∧*su* ∈ *Data-type-definition* **then** *su.fullQualifier1*
else *su.fullQualifier1*⌒**mk**-*Qualifier*(*su.entityKind1*, *su.entityName1*)
endif
endlet

idKind1(*id*:*Identifier*):*EntityKind1*=def
case *id.parentAS1* **of**
| *Create-request-node*∪*Signal-destination*: **agent**
| *Agent-type-definition*∪*Agent-definition*: **agent type**
| *Procedure-definition*∪*Call-node*∪*Value-returning-call-node*: **procedure**
| *Gate-definition*∪*Channel-path*∪*Output-node*∪*Save-signalset*: **signal**
| *Data-type-definition*∪*Parameter*∪*Result*∪*Signal-definition*∪*Timer-definition*∪
Formal-argument∪*Variable-definition*∪*Any-expression*: **sort**
| *Set-node*∪*Reset-node*∪*Timer-active-expression*: **timer**
| *Originating-gate*∪*Destination-gate*: **gate**
| *Composite-state-type-definition*∪*State-machine*∪*State-node*∪*State-partition*: **state type**
| *Literal*: **literal**
| *Open-range*∪*Operation-application*: **operation**
| *Input-node*:
if *id* **in** *id.parentAS1*.**s**-*Variable-identifier*-**seq then variable**
else signal
endif
| *Variable-access* ∪ *Assignment*: **variable**
| *Direct-via*:
if *getEntityDefinition1*(*id*, **channel**) ≠ *undefined* **then channel**
else gate

```
        endif
    endcase
```

The function *entityName1* is used to get the entity name for an entity definition in AS1.

```
entityName1(d: ENTITYDEFINITION1): Name =def
    case d of
    | Package-definition => d.s-Package-name
    | Agent-definition => d.s-Agent-name
    | Agent-type-definition => d.s-Agent-type-name
    | Procedure-definition => d.s-Procedure-name
    | State-node => d.s-State-name
    | Composite-state-type-definition => d.s-State-type-name
    | Channel-definition => d.s-Channel-name
    | Gate-definition => d.s-Gate-name
    | Signal-definition => d.s-Signal-name
    | Timer-definition => d.s-Timer-name
    | Variable-definition => d.s-Variable-name
    | Value-data-type-definition => d.s-Sort
    | Syntype-definition => d.s-Syntype-name
    | Interface-definition => d.s-Sort
    | Literal-signature => d.s-Literal-name
    | Operation-signature => d.s-Operation-name
    otherwise undefined
    endcase
```

The function *entityKind1* is used to get the entity kind for an entity definition on AS1.

```
entityKind1(d: ENTITYDEFINITION1): ENTITYKIND1 =def
    case d of
    | Package-definition => package
    | Agent-definition => agent
    | Agent-type-definition => agent type
    | Procedure-definition => procedure
    | State-node => state
    | Composite-state-type-definition => state type
    | Channel-definition => channel
    | Gate-definition => gate
    | Signal-definition => signal
    | Timer-definition => timer
    | Variable-definition => variable
    | Data-type-definition => sort
    | Syntype-definition => sort
    | Interface-definition => sort
    | Literal-signature => literal
    | Operation-signature => operation
    otherwise undefined
    endcase
```

The function *getEntityDefinition1* gets the entity definition for an identifier.

```
getEntityDefinition1(id: Identifier, k: ENTITYKIND1): ENTITYDEFINITION1=def
    take({d∈ENTITYDEFINITION1: d.identifier1 = id ∧ d.entityKind1 = k∧
            (d.entityKind1 = operation ⇒
                isActualAndFormalParameterMatched1(id.actualParameterListOfOpId1,
                                                   d.formalParameterSortList1))})
```

The function *agentKind1* is used to get the agent kind for an *Agent-definition* or *Agent-type-definition*.

```
agentKind1(d: Agent-definition∪Agent-type-definition):AGENTKIND1=def
    if d∈Agent-type-definition then d.s-Agent-kind
    else
        let td=getEntityDefinition1(d.s-Agent-type-identifier, agent type) in
            td.s-Agent-kind
```

```
        endlet
    endif
```

value1 returns a member of *VALUE1* corresponding to a *Sort*.

```
value1(e: Constant-expression):VALUE1 =def
    case e of
    | Literal(*) => computeLiteral(e)
    | Conditional-expression(bool, consequence, alternative) =>
        if bool.value1.semvalueBool then consequence.value1 else alternative.value1 endif
    | Equality-expression(a, b) => computeEquality(a.value1, b.value1)
    | Operation-application(proc, values) => computeConstant(proc, < v in values: (v.value1) >
    | Range-check-expression(range, expr) => expr.value1 ∈ range.range1
    endcase
```

computeLiteral returns the *Literal-signature* corresponding to the literal.

```
computeLiteral(id: Literal-identifier): VALUE1 =def
    getEntityDefinition1(id, idKind1(id) ).s-Literal-signature
```

semvalueBool returns the *BOOLEAN* value true if the *Literal-signature* is the Boolean value true, otherwise it returns the *BOOLEAN* value false.

```
semvalueBool (b: Literal-signature): BOOLEAN =def
    if b= mk-Literal-signature (
        mk-Name("true"),
        mk-Result(mk-Identifier(
            < mk-Package-qualifier(mk-Name("Predefined"))>, mk-Name("Boolean"))),
        undefined)
    then true else false endif
```

computeEquality returns the Boolean *Literal-signature* for true if the value of both expressions is the same, and the Boolean *Literal-signature* for false otherwise.

```
computeEquality (a: Literal-signature, b: Literal-signature): Literal-signature =def
    mk-Literal-signature (
        if a = b then mk-Name("true") else mk-Name("false") endif,
        mk-Result(mk-Identifier(
            < mk-Package-qualifier(mk-Name("Predefined"))>, mk-Name("Boolean")))
        undefined)
```

computeConstant returns the *Literal-signature* for the value of applying the operation to the constant expression argument values for the operation.

```
computeConstant (v: VALUE1-set): Literal-signature =def
    // Further study is needed to formulate this function.
    //The dynamic compute function of F3.3.1 could be used as a basis.
```

range1 returns the set of elements that satisfy the condition.

```
range1 (r: Range-condition): VALUE1-set =def
    { v∈VALUE1:
        ∃ci ∈ r.s-Condition-item:
            ((ci ∈ Open-range) ∧ isInOpenRange1(v, ci)) ∨
            ((ci∈Closed-range) ∧
                (∃s1, s2∈ ci.s-Open-range: (s1 ≠ s2) ∧ isInOpenRange1(v, s1)∧ isInOpenRange1(v, s2)))}

isInOpenRange1(v1: VALUE1, r: Open-range): BOOLEAN =def
    let operator = r.s-Operation-identifier.s-Name in
        let v' = r.s-Constant-expression.value0 in
            let v= v1.value0 in
            case operator of
            | "=" => v = v'
```

```
            | "/=" => ¬ (v = v')
            | "<=" => v ≤ v'
            | ">=" => v ≥ v'
            | ">" => v > v'
            | "<" => v < v'
            endcase
            endlet
        endlet
    endlet
```

The function *staticSort1* returns the static sort of *e*.

```
staticSort1(e: Expression ): Sort-reference-identifier =def
    case e of
    | Literal => getEntityDefinition1(e.s-Literal-identifier, literal).s-Result
    | Variable-identifier => getEntityDefinition1(e, variable).s-Sort-reference-identifier
    | Equality-expression∪Range-check-expression∪Timer-active-expression =>
        mk-Identifier(mk-Qualifier("Predefined"), "Boolean")
    | Now-expression => mk-Identifier(mk-Qualifier("Predefined"), "Time")
    | Pid-expression => mk-Identifier(mk-Qualifier("Predefined"), "Pid")
    | Any-expression => e.s-Sort-reference-identifier
    | Operation-application => getEntityDefinition1(e.s-Operation-identifier, operation).s-Result
    | Value-returning-call-node => getEntityDefinition1(e.s-Procedure-identifier, procedure).s-Result
    | Conditional-expression => staticSort1(e.s-Consequence-expression)
    otherwise undefined
    endcase
```

The predicate *isDirectSuperType1* is used to determine if the first entity definition is the direct super type of the second one.

```
isDirectSuperType1(d: ENTITYDEFINITION1, d': ENTITYDEFINITION1): BOOLEAN=def
    case d' of
    | Agent-type-definition =>
        d ∈Agent-type-definition ∧ d = getEntityDefinition1(d'.s-Agent-type-identifier, agent type)
    | Procedure-definition =>
        d∈Procedure-definition ∧ d = getEntityDefinition1(d'.s-Procedure-identifier, procedure)
    | Composite-state-type-definition =>
        d∈Composite-state-type-definition ∧
        d = getEntityDefinition1(d'.s-Composite-state-type-identifier, state type)
    | Value-data-type-definition =>
        d∈Value-data-type-definition ∧ d = getEntityDefinition1(d'.s-Data-type-identifier, sort)
    | Interface-definition =>
        d∈Interface-definition ∧
        (∃dataId∈Data-type-identifier: dataId .parentAS1= d' ∧ d = getEntityDefinition1(dataId, sort))
    | Syntype-definition =>
        isDirectSuperType1(d,d'.derivedDataType1)
    otherwise false
    endcase
```

The predicate *isSuperType1* is used to determine if the first entity definition is the super type of the second one.

```
isSuperType1(d: ENTITYDEFINITION1, d': ENTITYDEFINITION1): BOOLEAN =def
    isDirectSuperType1(d, d')∨∃ d''∈ENTITYDEFINITION1: isSuperType1 (d, d'')∧isSuperType1 (d'', d')
```

The function *derivedDataType1* is used to get the data type definition for a given *Syntype-definition*.

```
derivedDataType1(d: Syntype-definition∪Data-type-definition): Data-type-definition =def
    if d∈Data-type-definition then d
    else getEntityDefinition1(d.s-Parent-sort-identifier, sort).derivedDataType1
```

The function *isCompatibleTo1* determines if a *Sort-reference-identifier* is compatible to the other.

isCompatibleTo1(*id1*: *Sort-reference-identifier*, *id2*: *Sort-reference-identifier*): *BOOLEAN*=$_{\text{def}}$
let *d1 = getEntityDefinition1*(*id1*, **sort**) **in**
let *d2 = getEntityDefinition1*(*id2*, **sort**) **in**
d1=d2∨ *isSuperType1*(*d2*, *d1*)
endlet
endlet

F2.2.1.6.5 Function definitions on AS0

The function *name0* gets the name of a given entity or reference.

name0(*d*: *ENTITYDEFINITION0*∪ *REFERENCE0*): <name> =$_{\text{def}}$
case *d* **of**
| *REFERENCE0* => *d.referenceName0*
| *ENTITYDEFINITION0* => *d.entityName0*
otherwise *undefined*
endcase

The function *entityName0* gets the name of a given entity definition.

entityName0(*ed*: *ENTITYDEFINITION0*): <name> =$_{\text{def}}$
case *ed* **of**
| <package definition> => *ed*.**s-**<package heading>.**s-**<name>
| <system definition> => *ed*.**s-**<system heading>.**s-**<name>
| <block definition> => *ed*.**s-**<block heading>.**s-**<name>
| <process definition> => *ed*.**s-**<process heading>.**s-**<name>
| <external procedure definition> => *ed*.**s-**<procedure heading>.**s-**<name>
| <internal procedure definition> => *ed*.**s-**<procedure heading>.**s-**<name>
| <system type definition> => *ed*.**s-**<system type heading>.**s-**<name>
| <block type definition> => *ed*.**s-**<block type heading>.**s-**<name>
| <process type definition> => *ed*.**s-**<process type heading>.**s-**<name>
| <composite state type definition> =>*ed*.**s-**<composite state type heading>.**s-**<name>
| <textual gate definition> => *ed*.**s-**<name>
| <textual typebased system definition> => *ed*.**s-**<name>
| <textual typebased block definition> => *ed*.**s-**<name>
| <textual typebased process definition> => *ed*.**s-**<name>
| <textual typebased state partition def> => *ed*.**s-**<name>
| <composite state> => *ed*.**s-**<composite state heading>.**s-**<name>
| <data type definition> => *ed*.**s-**<data type heading>.**s-**<name>
| <signal definition> => *ed*.**s-**<name>
| <timer definition item> => *ed*.**s-**<name>
| <signal list definition> => *ed*.**s-**<name>
| <interface definition> => *ed*.**s-**<interface heading>.**s-**<name>
| <literal signature> =>
if *ed*∈<literal name> **then** *ed*
elseif *ed*∈<named number> **then** *ed*.**s-**<literal name>
else *undefined*
endif
| <operation signature> =>
if *ed*.**s-implicit** ∈<operation name> **then** *ed*.**s-implicit**
else *ed*.**s-implicit**.**s-**<operation<name>
endif
| <operation definition> => *ed*.**s-**<operation heading>.**s-**<name>
| *FORMALCONTEXTPARAMETER0*=> *ed*.**s-**<name>
| <syntype definition> =>
let *s=ed*.**s-implicit in**
if *s*∈<syntype definition syntype> **then** *s*.**s-**<syntype<name>
else *s*.**s-**<data type heading>.**s-**<data type<name>
endif
endlet endcase

The function *referenceName0* gets the name of a given reference.

referenceName0(*ref*: *REFERENCE0*): <name> =$_{def}$
case *ref* **of**
| *TYPEREFERENCE0*=>*ref*. **s-**<identifier>.**s-**<name>
| <operation reference> => *ref*.**s-**<operation heading>.**s-**<operation name>
otherwise *ref*.**s-**<name>
endcase

The function *kind0* gets the name of a given entity definition or reference.

kind0(*d*: *ENTITYDEFINITION0*∪ *REFERENCE0*): *ENTITYKIND0* =$_{def}$
case *ed* **of**
| *REFERENCE0* => *d.referenceKind0*
| *ENTITYDEFINITION0*=> *d*. *entityKind0*
endcase

The function *entityKind0* gets the name of a given entity definition.

entityKind0(*ed*: *ENTITYDEFINITION0*): *ENTITYKIND0* =$_{def}$
case *ed* **of**
| <package definition> => **package**
| <system definition> ∪ <textual typebased system definition> => **system**
| <block definition> ∪ <textual typebased block definition> => **block**
| <process definition> ∪ <textual typebased process definition> => **process**
| <system type definition> => **system type**
| <block type definition> => **block type**
| <process type definition> => **process type**
| <composite state> ∪ <textual typebased state partition def> => **state**
| <composite state type definition> ∪ <compositestate type context parameter> => **state type**
| <procedure definition> ∪ <procedure context parameter> => **procedure**
| <data type definition> ∪ <signal context parameter name> => **signal**
| <data type definition> ∪ <syntype definition> ∪ <sort context parameter> => **type**
| <operation definition> => *ed*.**s-**<operation heading>.**s-**<operation kind>
| <operation signature> =>
if *ed.parentAS0* ∈<operator list> **then operator else method endif**
| <interface definition> ∪ <interface context parameter name> => **interface**
| <textual gate definition> ∪ <gate context parameter> => **gate**
| <timer definition item> ∪ <timer context parameter name> => **timer**
| <signal list definition> => **signallist**
| <literal signature> => **literal**
| <agent type context parameter> ∪ <agent context parameter>=> *ed*.<agent kind>
| <remote procedure definition> ∪ <remote procedure context parameter> => **remote procedure**
| <synonym definition> ∪ <synonym context parameter name> => **synonym**
| <variable context parameter names> ∪ <variables of sort>=>**variable**
| <remotevariable contextparameter names> ∪ <remote variables of sort> =>
remote variable
otherwise *undefined*
endcase

The function *referenceKind0* gets the entity kind of a specified reference.

referenceKind0(*ref*: *REFERENCE0*): *ENTITYKIND0*=$_{def}$
case *ref* **of**
| <system type reference> => **system type**
| <block type reference> => **block type**
| <process type reference> => **process type**
| <composite state type reference> => **state type**
| <block reference> => **block**
| <process reference> => **process**
| <composite state reference> => **state**
| <package reference> => **package**
| <procedure reference> => **procedure**
| <operation reference> => *ref*.**s-**<operation kind>

otherwise *undefined*
endcase

The function *qualifier0* gets the qualifier specified in an entity definition or a reference.

qualifier0(*d*: *ENTITYDEFINITION0*∪*REFERENCE0*): <qualifier> =def
take({*q*∈<qualifier>: *q.parentAS0*. *parentAS0* = *d*})

The function *surroundingScopeUnit0* gets the surrounding scope unit for a node in AS0.

surroundingScopeUnit0(*d*: *DefinitionAS0*): *SCOPEUNIT0* =def
if *d* ∈ <referenced definition> **then**
parentAS0ofKind(*d.referencedBy0*, *SCOPEUNIT0*)
else
parentAS0ofKind(*d*, *SCOPEUNIT0*)
endif

F2.2.1.6.6 Lexis

The following lexical items are used here:

Keywords ... (implicitly as <keyword> :: ())

<plus sign> :: ()

<hyphen> :: ()

<greater than sign> :: ()

<greater than or equals sign> :: ()

<less than sign> :: ()

<less than or equals sign> :: ()

<equals sign> :: ()

<not equals sign> :: ()

<concatenation sign> :: ()

<implies sign> :: ()

<asterisk> :: ()

<solidus> :: ()

<infix operation name> ::
<implies sign> | **or** | **xor** | **and**
| <greater than sign> | <greater than or equals sign>
| <less than sign> | <less than or equals sign> | **in**
| <plus sign> | <hyphen> | <concatenation sign> | <asterisk> | <solidus> | **mod** | **rem**

If an <actual parameter> is omitted in the <actual parameter list> of <actual parameters>, this is represented as <undefined> in AS0.

<undefined> :: ()

F2.2.2 Visibility rules, names and identifiers

F2.2.2.1 Name

Abstract syntax

Name	::	*TOKEN*
Package-name	=	*Name*
Agent-type-name	=	*Name*
Agent-name	=	*Name*
State-type-name	=	*Name*

State-name = *Name*

Data-type-name = *Name*

Procedure-name = *Name*

Signal-name = *Name*

Interface-name = *Name*

Literal-name = *Name*

Operation-name = *Name*

Syntype-name = *Name*

Timer-name = *Name*

Gate-name = *Name*

Exception-name = *Name*

Connector-name = *Name*

State-entry-point-name = *Name*

State-exit-point-name = *Name*

Channel-name = *Name*

Variable-name = *Name*

NOTE – There are no concrete constructs in SDL-2010 for the use of exceptions, but the concept of an *Exception-name* and *Exception-identifier* is retained for descriptions of when an exception is raised. Therefore, an implementation that handles exceptions can extend the formal semantics. Similarly, for the user of the language there is no syntax for an exception name, but <exception name> is defined for the description of predefined data types [ITU-T Z.104] and the exception names are listed as part of **package** Predefined.

Concrete syntax

<name> :: *TOKEN*

<quoted operation name> :: *TOKEN*

<character string> :: *TOKEN*

<hex string> :: *TOKEN*

<bit string> :: *TOKEN*

<operation name> = <operator<name> | <quoted operation name>

<literal name> = <literal<name> | <string name>

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

NOTE – A lexical distinction is made (in clause 6.1 of [ITU-T Z.101]) between a <name>, an <integer name> and a <real name> to avoid some lexical ambiguities, whereas in the static formal semantics these are (currently) all treated as <name>.

Mapping to abstract syntax

```
| <name>(x) => mk-Name(x)
| <quoted operation name>(x) => mk-Name(x)
| <hex string>(x) => mk-Name(x)
| <bit string>(x) => mk-Name(x)
| <character string>(x) =>
    let cscontext = x.parentAS0 in
        case cscontext of
            | <transition option> ∪ <task> ∪ <decision>
                => mk-Informal-text(x)
            | <textual answer part>
                =>
                let q = cscontext.parentAS0.parentAS0.s-<question> in
```

```
            if q=<character string> then mk-Informal-text(x)
            else
                if   (isSubSort0(<identifier>(<qualifier>(< <path item>(package,<name>(
                        "Predefined")) >),<name>("Charstring")), q.getStaticSort0)) ∨
                    (length(x) =1) ∧
                    (isSubSort0(<identifier>(<qualifier>(< <path item>(package,<name>(
                        "Predefined")) >),<name>("Char"),) q.getStaticSort0)) ∨
                    (isSubSort0(<identifier>(<qualifier>(< <path item>(package,<name>(
                        "Predefined")) >),<name>("NumericString")), q.getStaticSort0)) ∨
                    (isSubSort0(<identifier>(<qualifier>(< <path item>(package,<name>(
                        "Predefined")) >),<name>("PrintableString")), q.getStaticSort0)) ∨
                    (isSubSort0(<identifier>(<qualifier>(< <path item>(package,<name>(
                        "Predefined")) >),<name>("TeletexString")), q.getStaticSort0)) ∨
                    (isSubSort0(<identifier>(<qualifier>(< <path item>(package,<name>(
                        "Predefined")) >),<name>("VideotexString")), q.getStaticSort0)) ∨
                    (isSubSort0(<identifier>(<qualifier>(< <path item>(package,<name>(
                        "Predefined")) >),<name>("VisibleString")), q.getStaticSort0))
                then mk-Name(x)
                else mk-Informal-text(x)
                endif
            endif
        endlet
    otherwise mk-Name(x)
  endcase
endlet
```

A <character string> in a <textual answer part> is a special case. If the <question> is informal, all the answers have to be <character string> answers and are informal. If the question is an expression (but not a <character string>) of a sort with values that have a <character string> representation (as detailed above), a <character string> represents a value of the sort. If the sort of the question expression is not one of these sorts, the <character string> answer is informal.

F2.2.2.2 Identifier

Abstract syntax

Identifier	::	*Qualifier Name*
Qualifier	=	*Path-item+*
Agent-identifier	=	*Identifier*
Agent-type-identifier	=	*Identifier*
Procedure-identifier	=	*Identifier*
Signal-identifier	=	*Identifier*
Data-type-identifier	=	*Identifier*
Sort-reference-identifier	=	*Sort-identifier*
	|	*Syntype-identifier*
Sort-identifier	=	*Identifier*
Syntype-identifier	=	*Identifier*
Timer-identifier	=	*Identifier*
Gate-identifier	=	*Identifier*
Exception-identifier	=	*Identifier*
Composite-state-type-identifier	=	*Identifier*
Literal-identifier	=	*Identifier*
Operation-identifier	=	*Identifier*
Variable-identifier	=	*Identifier*

Conditions on abstract syntax

$\forall d, d' \in$ *EntityDefinition1*: *d.entityKind1 = d'.entityKind1* $\wedge$ *d* $\neq$ *d'* $\Rightarrow$ *d.identifier1* $\neq$ *d'.identifier1*

All entities with the same entity kind must have different *Identifiers*.

Concrete syntax

<identifier> :: <qualifier> <name>

<qualifier> = <path item>*

Conditions on concrete syntax

$\forall$*def1, def2*$\in$*EntityDefinition0*:
(*def1* $\notin$<operation signature> $\wedge$ *def2* $\notin$ <operation signature> $\wedge$ *def1* $\neq$ *def2* $\wedge$
def1.surroundingScopeUnit0 = def2.surroundingScopeUnit0 $\wedge$
def1.entityKind0= def2.entityKind0) $\Rightarrow$
def1.entityName0$\neq$ *def2.entityName0*

No two definitions in the same scope unit and belonging to the same entity kind can have the same <name>. The only exceptions are operations defined in the same <data type definition>, as long as they differ in at least one argument <sort> or the result <sort>.

Transformations

i=<identifier>(*, *) =12=>
let *full* = *fullIdentifier0*(*i*) **in if** *i* = *full* **then** *i* **else** *full* **endif endlet**

Mapping to abstract syntax

| <identifier>(*q*, *name*) => **mk-***Identifier*(*Mapping*(*q*), *Mapping*(*name*))

Auxiliary functions

For any given identifier, return its full identifier.

fullIdentifier0(*i*:<identifier>): <identifier> =$_{\text{def}}$ *i.refersto0.identifier0*

For any given identifier, return the definition it refers to.

refersto0(*i*:<identifier>): *EntityDefinition0* =$_{\text{def}}$ *getEntityDefinition0*(*i*, *idKind0*(*i*))

For any given entity definition in AS0, the function *identifier0* returns its identifier with full qualifier.

identifier0(*def*: *EntityDefinition0*): <identifier> =$_{\text{def}}$
<identifier>(*def.fullQualifier0*, *def.entityName0*)

The function *fullQualifier0* is used to get the full qualifier for an entity definition.

fullQualifier0(*d*: *EntityDefinition0*): <qualifier> =$_{\text{def}}$
let *su* = *d*. *surroundingScopeUnit0* **in**
if *su* = *undefined* **then** *empty*
else *su.fullQualifier0*$^\frown$<path item>(*su.entityKind0*, *su.entityName0*)
endlet

The function *getEntityDefinition0* is used to get the definition that the given identifier refers to.

getEntityDefinition0(*id*:<identifier>, *ek*: *EntityKind0*): *EntityDefinition0* =$_{\text{def}}$
if *ek*$\in${**operator**, **literal**, **method**} **then**
resolutionByContext0(*id*)
else
let *su* = *getStartingScopeUnit0*(*id*, *id.surroundingScopeUnit0*) **in**
if *su* = *undefined* **then** *undefined*
else *resolutionByContainer0*(*su*, *id*, *ek*)

```
            endif
        endlet
    endif
```

The function *resolutionByContainer0* binds an <identifier> to a definition through resolution by container.

```
resolutionByContainer0(su: SCOPEUNIT0, id:<identifier>, ek:ENTITYKIND0): ENTITYDEFINITION0 =def
    let d1=bindInLocalDefinition0(su, id, ek) in
        if d1≠ undefined then d1
        else let d2=bindInBaseType0(su, id, ek) in
            if d2≠ undefined then d2
            else let d3=bindInUsedPackage0(su.s-<package use clause>, id, ek) in
                if d3≠ undefined then d3
                else let d4=bindInLocalInterface0(su.localInterfaceDefinitionSet0, id, ek) in
                    if d4≠ undefined then d4
                    else let su'=su.surroundingScopeUnit0 in
                        if su' ≠ undefined then resolutionByContainer0(su', id, ek)
                        else undefined
                        endif endlet
                    endif endlet
                endif endlet
            endif endlet
        endif endlet
```

The function *bindInLocalDefinition0* is used to search in the given scope unit to determine if there exists a local entity definition for the specified identifier.

```
bindInLocalDefinition0(su: SCOPEUNIT0, id: <identifier>, ek: ENTITYKIND0): ENTITYDEFINITION0=def
    let d = take({d∈ ENTITYDEFINITION0:
            d.surroundingScopeUnit0 = su ∧ isSameEntityName0(id.s-<name>, d )∧
            isConsistentKindTo0(d.entityKind0, ek)∧isVisibleIn0(d, id.surroundingScopeUnit0)}) in
        if d ≠ undefined then d
        else let rd = take({rd ∈REFERENCE0 : rd.surroundingScopeUnit0 = su ∧
                rd. referencedDefinition0.entityName0= id.s-<name>∧
                isConsistentKindTo0(rd.referencedDefinition0.entityKind0, ek) ∧
                isVisibleIn0(rd, id.surroundingScopeUnit0))}) in
            if rd ≠ undefined then rd.referencedDefinition0
            else undefined
            endif endlet
        endif
    endlet
```

The function *bindInBaseType0* finds the entity definition corresponding to the given <identifier> in the base type of the scope unit.

```
bindInBaseType0(su: SCOPEUNIT0, id: <identifier>, ek: ENTITYKIND0): ENTITYDEFINITION0=def
    let spec = su.specialization0 in
        if (spec = undefined) ∨ isAncestorAS0(spec, id) then undefined
        else resolutionByContainer0(spec.s-<type expression>.baseType0, id, ek)
        endif
    endlet
```

The function *bindInUsedPackage0* finds the entity definition corresponding to the given <identifier> in the used packages of the scope unit.

```
bindInUsedPackage0(ucl:<package use clause>*, id: <identifier>, ek: ENTITYKIND0):
    ENTITYDEFINITION0=def
    if ucl = empty then undefined
    elseif ucl.head = id.parentAS0 then
        bindInUsedPackage0(ucl.tail, id, ek)
    else
```

```
        let d = bindInLocalDefinition0(ucl.head.usedPackage0, id, ek) in
            if d ≠ undefined then d
            else bindInUsedPackage0(ucl.tail, id,ek)
            endif
        endlet
    endif
```

The function *bindInLocalInterface0* finds the entity definition corresponding to the given <identifier> in the interfaces of the scope unit.

```
bindInLocalInterface0(is:<interface definition>-set, id: <identifier>, ek: EntityKind0):
    EntityDefinition0 =def
    if is = ∅ then undefined
    else let d = is.take in
            let ed = bindInLocalDefinition0(d, id, ek) in
        if ed ≠ undefined then ed
        else bindInLocalInterface0(is \ {d})
        endif
        endlet
    endif
```

The function *isSameEntityName0* is used to determine if the given name has the same name as the entity definition.

```
isSameEntityName0(n: <name>,d: EntityDefinition0 ): Boolean=def
    (n = d.entityName0)
```

For a given identifier (left most path item may be omitted), the function *getStartingScopeUnit0* gets the starting scope unit denoted by the partial qualifier.

```
getStartingScopeUnit0(id: <identifier>, su: ScopeUnit0): ScopeUnit0=def
    if su= undefined then undefined
    elseif isQualifierMatched0(id.s-<qualifier>, su) then su
    else let su1= getStartingSuInUsedPackage0(id, su.usedPackageDefinitionList0) in
        if su1≠ undefined then su1
        else let su2= getStartingSuInInterface0(id, su.localInterfaceDefinitionSet0) in
            if su2≠ undefined then su2
            else getStartingScopeUnit0(id, su.surroundingScopeUnit0)
            endif
            endlet
        endif endlet
    endif
```

The function *getStartingSuInUsedPackage0* finds the starting scope unit in the packages list.

```
getStartingSuInUsedPackage0(id: <identifier>, pdl:<package definition>*): ScopeUnit0=def
    if pdl =empty then undefined
    elseif isQualifierMatched0(id.s-<qualifier>, pdl.head) then pdl.head
    else getStartingSuInUsedPackage0(id, pdl.tail)
    endif
```

The function *getStartingSuInInterface0* finds the starting scope unit in the interface list.

```
getStartingSuInInterface0(id: <identifier>, ifds:<interface definition>-set): ScopeUnit0=def
    if ifds = ∅ then undefined
    else let d = ifds.take in
        if isQualifierMatched0(id.s-<qualifier>, d) then d
        else getStartingSuInInterface0(id, ifds \ {d})
        endif endlet
    endif
```

The function *isDefinedIn0* determines if an entity definition is defined in a given scope unit.

isDefinedIn0(*ed*: *ENTITYDEFINITION0*, *su*: *SCOPEUNIT0*): *BOOLEAN*=$_{\text{def}}$
if (*su* = *undefined* ∨ *ed* =*undefined*)**then** *false*
else
let *su'* = *ed.surroundingScopeUnit0* **in**
su = *su'* ∨
isVisibleInInterface0(*ed*, *su*) ∨
isVisibleInDataType0(*ed*, *su*) ∨
isVisibleThroughBaseType0(*ed*, *su*)
endlet
endif

The function *isVisibleIn0* determines if an entity definition is visible in a given scope unit.

isVisibleIn0(*ed*: *ENTITYDEFINITION0*, *su*: *SCOPEUNIT0*): *BOOLEAN*=$_{\text{def}}$
isDefinedIn0(*ed*, *su*) ∨
isVisibleThroughUsedPackage0(*ed*, *su*) ∨
isVisibleIn0(*ed*, *su.surroundingScopeUnit0*)

The function *isVisibleInInterface0* determines if the scope unit contains an <interface definition> which is the defining context of the entity.

isVisibleInInterface0(*ed*: *ENTITYDEFINITION0*, *su*: *SCOPEUNIT0*): *BOOLEAN*=$_{\text{def}}$
(*ed.surroundingScopeUnit0*∈ *su.localInterfaceDefinitionSet0*)

The function *isVisibleInDataType0* determines if the scope unit contains a <data type definition> which is the defining context of the entity.

isVisibleInDataType0(*ed*: *ENTITYDEFINITION0*, *su*: *SCOPEUNIT0*): *BOOLEAN*=$_{\text{def}}$
(*ed.surroundingScopeUnit0*∈ *su.localDataTypeDefinitionSet0*) ∧
(*ed*∈<literal signature> ∪ <operation signature> ⇒ *ed.isPublic0*)

The function *isVisibleThroughBaseType0* determines if the entity is visible through the base type of the scope unit.

isVisibleThroughBaseType0(*ed*: *ENTITYDEFINITION0*, *su*: *SCOPEUNIT0*): *BOOLEAN*=$_{\text{def}}$
let *spec* = *su*. *specialization0* **in**
if *spec* = *undefined* **then** *false*
else (∃*btd*∈*TYPEDEFINITION0* : *isDirectSubType0*(*su*, *btd*) ∧ *isVisibleIn0*(*ed*, *btd*) ∧
(*ed* ∈ <literal signature>∪<operation signature> ⇒
¬*isPrivate0*(*ed*)∧¬ *isRenamedBy0*(*ed*, *spec*))∧
ed∈ *FORMALCONTEXTPARAMETER0*⇒¬*isBoundToActualContextPara0*(*ed*, *spec*))
endif
endlet

The function *isVisibleThroughUsedPackage0* determines if an entity definition is visible through used packages.

isVisibleThroughUsedPackage0(*ed*: *ENTITYDEFINITION0*, *su*: *SCOPEUNIT0*): *BOOLEAN* =$_{\text{def}}$
let *ucs* = *su*.**s-**<package use clause>.*toSet* **in**
if *ucs* = ∅ **then** *false*
else (∃*uc*∈*ucs*: *ed.surroundingScopeUnit0*= *uc.usedPackage0*∧
isVisibleThroughUseClause0(*ed*, *uc*))
endif
endlet

The function *isBoundToActualContextPara0* determines if a <formal context parameter> is bound to an <actual context parameter> in a <specification>.

isBoundToActualContextPara0 (*fcp*: *FORMALCONTEXTPARAMETER0*, *spec*: <specialization>):
BOOLEAN =$_{\text{def}}$
(∃*acp*∈<actual context parameter>:
acp. *parentAS0.parentAS0* = *spec*∧*isContextParameterCorresponded0*(*fcp*, *acp*))

The function *isVisibleThroughUseClause0* determines if an entity definition is visible through the <package public> and <package use clause>.

```
isVisibleThroughUseClause0(ed: ENTITYDEFINITION0, uc:<package use clause>): BOOLEAN =def
    let pd=uc.usedPackage0 in
    let pi = pd. s-<package heading>.s-<package public> in
        (pi = undefined∨
            ∃ds∈ <definition selection>: ds.parentAS0= pi∧
                isMentionedInDefSelection0(ed, ds, pd)) ∧
        (uc.s-<definition selection>-seq = empty ∨
            ∃ds∈<definition selection>: ds.parentAS0 = uc ∧
                isMentionedInDefSelection0(ed, ds, pd))
    endlet
```

The function *isMentionedInDefSelection0* determines if an entity is mentioned in a <definition selection>.

```
isMentionedInDefSelection0
    (ed: ENTITYDEFINITION0, ds:<definition selection>, pd:<package definition>):
        BOOLEAN =def
    (ds.s-<name> = ed. entityName0 ∧
        (ds.s-<selected entity kind> ≠undefined⇒ ds.s-<selected entity kind> = ed. entityKind0))∨
    (ed.entityKind0= signal ∧ ds.s-<selected entity kind> = signallist ∧
        (∃sld∈<signal list definition>: sld.surroundingScopeUnit0= pd∧
            sld. entityName0= ds.s-<name> ∧
            (∃sigId∈ sld.signalSet0: getEntityDefinition0(sigId, signal) = ed)))
```

The function *isConsistentKindTo0* is used to determine if the first entity kind is consistent to the second one.

```
isConsistentKindTo0(t1: ENTITYKIND0, t2: ENTITYKIND0): BOOLEAN =def
    t1 = t2 ∨
    (t2= agent) ∧ ((t1 = system)∨(t1 = block)∨(t1 = process)) ∨
    (t2= agent type) ∧ ((t1 = system type)∨(t1 = block type)∨(t1 = process type)) ∨
    (t2 = sort)∧(( t1=interface)∨( t1= type))
```

The function *isQualifierMatched0* is used to determine if the given <qualifier> is the same as the rightmost part of the full <qualifier> denoting the given scope unit.

```
isQualifierMatched0(q: <qualifier>, su: SCOPEUNIT0): BOOLEAN =def
    if q = undefined then true
    elseif su∈<compound statement> then false
    else let q' = su.fullQualifier0 in
        let seq1 = q.s-<path item>-seq in
        let seq2 = q'⌒<path item>(su.entityKind0, su.entityName0) in
            (seq1.length≤ seq2. length∧
                (∀i∈1.. seq1.length:∀j∈NAT: j = seq2.length - seq1. length+ i ⇒
                    (seq1[i].s-<name> = seq2[j].s-<name> ∧
                        (seq1[i].s-<scope unit kind> ≠ undefined ⇒
                            seq1[i].s-<scope unit kind> = seq2[j].s-<scope unit kind>))))
        endlet
    endif
```

The function *resolutionByContext0* is used to bind all <name>s of entity kind **operator**, **method** and **literal** to their corresponding entity definitions.

```
resolutionByContext0(id:<identifier>):ENTITYDEFINITION0=def
    let bl = take(getBindingListSet0(id.contextOfIdentifier0)) in
        getDefinitionInBindingList0 (id.s-<name>, bl)
    endlet
```

The function *contextOfIdentifier0* gets the context for resolving the identifier.

```
contextOfIdentifier0(id:<identifier>):CONTEXT0=def
    if (∃exp∈<expression>: isAncestorAS0(exp, id)) then
        contextOfExp0(parentAS0ofKind(id, <expression>))
    else undefined
```

The function *getBindingListSet0* is used to bind all <name>s of entity kind **operator**, **method** and **literal** in the context to their corresponding entity definitions.

```
getBindingListSet0(c: CONTEXT0): BINDINGLIST0-set =def
    let nameList = c.nameList0 in
    let possibleBindingListSet = nameList.possibleBindingListSet0 in
    let possibleResultSet = {pbl∈ possibleBindingListSet: isSatisfyStaticCondition0(pbl, c)} in
    let resultSet = {r∈ possibleResultSet: ∀r'∈ possibleResultSet: r ≠ r' ⇒
                        mismatchNumber0(r, c)≤ mismatchNumber0(r', c)} in
        if |resultSet| = 1 then resultSet
        elseif |resultSet| = 0 then ∅
        else ∅
        endif
    endlet
```

The function *nameList0* gets all the <name>s of entity kind **operator**, **method** and **literal** appearing in the context.

```
nameList0(c: CONTEXT0): <name>*=def
    case c of
    |<assignment>=>c.s-<variable>.nameListInVariable0⌒c.s-<expression>.nameListInExpression0
    |<decision>=>c.nameListInDecision0
    |<expression>=>c.nameListInExpression0
    otherwise empty
    endcase
```

The function *nameListInExpression0* gets all the <name>s of entity kind **operator**, **method** and **literal** appearing in the <expression>.

```
nameListInExpression0(exp: <expression>):<name>*=def
    case exp of
    |<create expression>∪<value returning procedure call>=>
        exp.actualParameterList0.nameListInActualParameterList0
    |<range check expression> => exp.s-<expression>.nameListInExpression0
    |<binary expression>=>
        let mk-<binary expression>(e1, op, e2)= exp in
            e1.nameListInExpression0⌒ op ⌒ e2.nameListInExpression0
        endlet
    |<equality expression>=>
        exp. s-<expression>.nameListInExpression0⌒exp.s2-<expression>.nameListInExpression0
    |<operand5>=>exp.s-implicit⌒exp.s-<primary>.nameListInPrimary0
    endcase
```

The function *nameListInPrimary0* gets all the <name>s of entity kind **operator**, **method** and **literal** appearing in the <primary>.

```
nameListInPrimary0(p: <primary>):<name>*=def
    case p of
    |<operator application>=>p.nameListInOperationApplication0
    |<literal>=><p.s-<literal identifier>.s-<literal name>>
    |<expression>=>p.nameListInExpression0
    |<conditional expression>=>
        p.s-<expression>.nameListInExpression0⌒
```

p.**s-**<consequence expression>.*nameListInExpression0*⌒
p.**s-**<alternative expression>.*nameListInExpression0*
|<spelling term>=><*p*.**s-** <name>>
|<extended primary>=>*p.nameListInExtendedPrimary0*
otherwise: *empty*
endcase

The function *nameListInOperationApplication0* gets all the <name>s of entity kind **operator**, **method** and **literal** appearing in the <operation application>.

nameListInOperationApplication0(*oa*: <operator application>):<name>*=$_{def}$
case *oa* **of**
|<operator application>=>
<*oa*.**s-**<operation identifier>>⌒*oa.actualParameterList0.nameListInActualParameterList0*
|<method application>=>
oa.**s-**<primary>.*nameListInPrimary0*⌒<*oa*.**s-**<operation identifier>.**s-**<operation name>>⌒
oa.actualParameterList0.nameListInActualParameterList0
endcase

The function *nameListInExtendedPrimary0* gets all the <name>s of entity kind **operator**, **method** and **literal** appearing in the <extended primary>.

nameListInExtendedPrimary0(*ep*:<extended primary>):<name>*= $_{def}$
case *ep* **of**
|<indexed primary>=>
ep.**s-**<primary>.*nameListInPrimary0*⌒
ep.**s-**<actual parameter>**-seq**.*nameListInActualParameterList0*
|<field primary>=>*ep*.**s-**<primary>.*nameListInPrimary0*
|<composite primary>=>*ep*.**s-**<actual parameter>**-seq**.*nameListInActualParameterList0*
endcase

The function *nameListInActualParameterList0* gets all the <name>s of entity kind **operator**, **method** and **literal** appearing in the actual parameter list.

nameListInActualParameterList0(*el*: <expression>*): <name>*=$_{def}$
if *el* = *empty* **then** *empty*
else
el.head.nameListInExpression0⌒*el.tail.nameListInActualParameterList0*
endif

The function *nameListInVariable0* gets all the <name>s of entity kind **operator**, **method** and **literal** appearing in <variable>.

nameListInVariable0(*v*:<variable>):<name>*=$_{def}$
if *v*∈<indexed variable> **then**
v.**s-**<variable>.*nameListInVariable0*⌒
v.**s-**<actual parameter>**-seq**.*nameListInActualParameterList0*
elseif *v*∈<field variable> **then**
v.**s-**<variable>.*nameListInVariable0*
else *empty*
endif

The function *nameListInDecision0* gets all the <name>s of entity kind **operator**, **method** and **literal** appearing in the <decision>.

nameListInDecision0 (*d*: <decision>): <name>*= $_{def}$
d.**s-**<question>.*nameListInExpression0*⌒ *d.rangeConditionList0.nameListInRangeConditions0*

The function *nameListInRangeConditions0* gets all the <name>s of entity kind **operator**, **method** and **literal** appearing in the <range condition> list.

nameListInRangeConditions0 (*rcl*: <range condition>*): <name>*= $_{def}$
if *rcl = empty* **then** *empty*
else *rcl.head.nameListInRangeList0*⌒*rcl.tail.nameListInRangeConditions0*

The function *nameListInRangeList0* gets all the <name>s of entity kind **operator**, **method** and **literal** appearing in the <range> list.

nameListInRangeList0 (*rl*: <range>*): <name>*= $_{def}$
if *rl = empty* **then** *empty*
else *rl.head.nameListInRange0*⌒*rl.tail.nameListInRangeList0*

The function *nameListInRange0* gets all the <name>s of entity kind **operator**, **method** and **literal** appearing in the <range>.

nameListInRange0(*r*: <range>): <name>*= $_{def}$
case *r* **of**
|<closed range>=>
let *r* = **mk-**<closed range> (*c1*, *c2*) **in**
c1.nameListInExpression0⌒*c2.nameListInExpression0*
endlet
|<open range>=>
let *r* = **mk-**<open range>(*c1*, *n*, *c2*) **in**
c1.nameListInExpression0⌒<*n*>⌒*c2.nameListInExpression0*
endlet
endcase

Each element in the *possibleBindingListSet0* represents a possible resolution for the given name list.

possibleBindingListSet0(*n*: <name>*):*BINDINGLIST0*-**set** =$_{def}$
{*b*∈*BINDINGLIST0*: *b.length* = *n.length*∧
∀*i*∈1.. *b.length*:*b*[*i*].**s-**<name>= *n*[*i*]∧ *b*[*i*].**s-***ENTITYDEFINITION0*∈*n*[*i*].*possibleDefinitionSet0*}

The function *isSatisfyStaticCondition0* determines if the binding violates any static sort constraints in the context.

isSatisfyStaticCondition0(*bl*:*BINDINGLIST0*, *c*:*CONTEXT0*): *BOOLEAN*=$_{def}$
case *c* **of**
|<assignment>=>*isSatisfyAssignmentCondition0*(*bl*, *c*)
|<decision>=>*isSatisfyDecisionCondition0*(*bl*, *c*)
|<expression>=>*isSatisfyExpressionCondition0*(*bl*, *c*)
otherwise *false*
endcase

The function *isSatisfyAssignmentCondition0* determines if the binding violates any static sort constraints in the <assignment>.

isSatisfyAssignmentCondition0(*bl*:*BINDINGLIST0*, *ass*: <assignment>): *BOOLEAN*=$_{def}$
let *varSort= getVariableSort0*(*ass*.**s-**<variable>) **in**
let *expSort= getStaticSort0*(*ass*.**s-**<expression>, *bl*) **in**
(*isSortCompatible0*(*varSort*, *expSort*) ∨ *isSortCompatible0*(*expSort*, *varSort*))∧
(*ass*.**s-**<variable>∈<indexed variable>⇒
isSatisfyIndexVariableCondition0(*bl*, *ass*.**s-**<variable>)
endlet

The function *isSatisfyIndexVariableCondition0* determines if the binding violates any static sort constraints in the <indexed variable>.

```
isSatisfyIndexVariableCondition0(bl, var:<indexed variable>): BOOLEAN=def
    let acp=var.s-<actual parameter>-seq in
        isSortCompatible0(getStaticSort0(acp[1],bl), getIndexSort0(getVariableSort0(var)))∧
        (var.s-<variable>∈<indexed variable>⇒
            isSatisfyIndexVariableCondition0(bl, var.s-<variable>))
    endlet
```

Get the static sort of a <variable>.

```
getVariableSort0(var:<variable>):<sort>=def
    case var of
    | <identifier>=>getEntityDefinition0(var, variable).parentAS1.s-<sort>
    | <indexed variable>=>getItemSort0(getVariableSort0(var.s-<variable>))
    | <field variable>=>getFieldSort0(getVariableSort0(var.s-<variable>), var.s-<field name>)
    endcase
```

The function *getItemSort0* gets the item sort of a <sort> that is a subtype of the predefined sort String or Array.

```
getItemSort0(s: <sort>):<sort>=def
    let d=getEntityDefinition0(s, sort).derivedDataType0 in
        if d.specialization0 = undefined then undefined
        elseif d.specialization0.s-<base type>.s-<name> = "String" then
            d.actualContextParameterList0[1]
        elseif d.specialization0.s-<base type>.s-<name> = "Array" then
            d.actualContextParameterList0[2]
        else undefined
        endif
    endlet
```

The function *getIndexSort0* gets the index sort of a <sort> that is a subtype of the predefined sort String or Array.

```
getIndexSort0(s: <sort>):<sort>=def
    let d=getEntityDefinition0(s, sort).derivedDataType0 in
        if d.specialization0 = undefined then undefined
        elseif d.specialization0.s-<base type>.s-<name> = "String" then
            <identifier>(<qualifier>(< <path item>(package,<name>("Predefined")) >),
                <name>("Integer"))
        elseif d.specialization0.s-<base type>.s-<name> = "Array" then
            d.actualContextParameterList0[1]
        else undefined
        endif
    endlet
```

The function *getFieldSort0* gets the field sort of a field name in the <data type definition> referred by the given <sort>.

```
getFieldSort0(s: <sort>, n:<name>):<sort>=def
    let d=getEntityDefinition0(s, sort).derivedDataType0 in
    let cons= d.s-<data type definition body>.s-<data type constructor> in
        if cons ∈ <structure definition> then
            take({fos.s-<sort>: fos∈<fields of sort>∧
                    (∃name∈<name>: name.parentAS0=fos∧name=n)})
        else undefined
        endif
    endlet
```

The function *isSatisfyStaticCondition0* determines if the binding violates any static sort constraints in the <decision>.

```
isSatisfyDecisionCondition0(bl:BINDINGLIST0, d: <decision>): BOOLEAN=def
    let q=d.s-<question>, rcs=d.rangeConditionList0.toSet in
```

```
        isSatisfyExpressionCondition0(bl, q)∧
        (∀rc1∈rcs:∀ce1∈<constant expression>:isAncestorAS1(rc1, ce1)⇒
            isSatisfyExpressionCondition0(bl, ce1)∧
            isSortCompatible0(getStaticSort0(ce1, bl), getStaticSort0(q, bl)∧
        (∀rc2∈rcs: ∀ce2∈<constant expression>:isAncestorAS1(rc2, ce2)⇒
            (isSortCompatible0(getStaticSort0(ce1, bl), getStaticSort0(ce2, bl))∨
                isSortCompatible0(getStaticSort0(ce2, bl), getStaticSort0(ce1, bl))))
    endlet
```

The function *isSatisfyExpressionCondition0* determines if the binding violates any static sort constraints in the <expression>.

```
isSatisfyExpressionCondition0(bl:BINDINGLIST0, exp: <expression>): BOOLEAN=def
    case exp of
    |<create expression>=>isSatisfyCreateCondition0(bl, exp)
    |<value returning procedure call>=>
        if exp∈<procedure call body> then isSatisfyProcedureCallBodyCondition0(bl, exp)
        else isSatisfyRemoteProcCallBodyCondition0(bl, exp)
        endif
    |<range check expression> => isSatisfyRangeCheckCondition0(bl, exp)
    |<equality expression>=> isSatisfyEqualityExpCondition0(bl, exp)
    |<binary expression>=>
        let opDef = getDefinitionInBindingList0(exp.s-implicit, bl) in
        let fpl = opDef.operationParameterSortList0 in
            fpl.length = 2∧
            isSortCompatible0(getStaticSort0(exp.s-<expression>, bl), fpl[1])∧
            isSortCompatible0(getStaticSort0(exp.s2-<expression>, bl), fpl[2])∧
            isSatisfyExpressionCondition0 (bl, exp.s-<expression>)∧
            isSatisfyExpressionCondition0 (bl, exp.s2-<expression>)
        endlet
        endlet
    |<operand5>=>
        if exp.s-implicit = undefined then
            isSatisfyPrimaryCondition0(bl, pr)
        else
            let opDef = getDefinitionInBindingList0(exp.s-implicit, bl) in
            let fpl = opDef.operationParameterSortList0 in
                fpl.length = 1∧isSortCompatible0(getStaticSort0(pr, bl), fpl[1])∧
                isSatisfyPrimaryCondition0(bl, pr)
            endlet
            endlet
        endif
    endcase
```

The function *isSatisfyCreateCondition0* determines if the binding violates any static sort constraints in the <expression>.

```
isSatisfyCreateCondition0(bl: BINDINGLIST0, ce: <create expression>):BOOLEAN =def
    let def = ce.getCreatedAgentDefinition0 in
    if def =undefined then false
    else
        let fpl = def.agentFormalParameterList0 in
        let apl = ce.actualParameterList0 in
            fpl.length=apl.length ∧
            (∀i∈1..fpl.length: isSortCompatible0(getStaticSort0(apl[i], bl), fpl[i].parentAS1.s-<sort>)∧
            isSatisfyExpressionCondition0(bl, apl[i]))
        endlet
        endlet
    endif endlet
```

The function *getCreateExpSort0* gets the static sort of <create expression>.

getCreateExpSort0 (*ce*: <create expression>): <sort>=$_{def}$
let *def = ce.getCreatedAgentDefinition0* **in**
if *def =undefined* **then** Pid
else *def.identifier0*
endif endlet

The function *isSatisfyProcedureCallBodyCondition0* determines if the binding violates any static sort constraints in the <procedure call body>.

isSatisfyProcedureCallBodyCondition0(*bl*: *BINDINGLIST0*, *body*: <procedure call body>):*BOOLEAN*=$_{def}$
let *apl = body.actualParameterList0* **in**
let *fpsl=body.calledProcedure0.formalParameterSortList0* **in**
fpsl.length=apl.length ∧
(∀*i*∈1..*fpsl.length*: *isSortCompatible0*(*getStaticSort0*(*apl*[*i*], *bl*), *fpsl*[*i*)∧
isSatisfyExpressionCondition0(*bl*, *apl*[*i*]))
endlet
endlet

The function *getProcCallBodySort0* gets the static sort of <procedure call body>.

getProcCallBodySort0 (*body*: <procedure call body>): <sort>=$_{def}$
case *body*.**s-implicit of**
| <identifier>=>*getEntityDefinition0*(*body*.**s-implicit**, **procedure**).*procedureResult0*
pd.**s-**<procedure heading>.**s-**<procedure result>
| <type expression>=> *body*.**s-implicit**.*baseType0.procedureResult0*
endcase

The function *procedureResult0* gets the result sort of a <procedure definition>.

procedureResult0(*pd*: <procedure definition>):<sort>=$_{def}$
if *pd*.**s-**<procedure heading>.**s-**<procedure result>≠*undefined* **then**
pd.**s-**<procedure heading>.**s-**<procedure result>
elseif *pd.specialization0* ≠ *undefined* **then**
pd.baseType0.procedureResult0
else *undefined*
endif

The function *getRemoteProcCallBodySort0* gets the static sort of <remote procedure call body>.

getRemoteProcCallBodySort0 (*body*: <remote procedure call body>, *bl*: *BINDINGLIST0*): <sort>=$_{def}$
getEntityDefinition0(*body*.**s-**<remote procedure<identifier>,**remote procedure**).*procedureResult0*

The function *isSatisfyRemoteProcCallBodyCondition0* determines if the binding violates any static sort constraints in the <remote procedure call body>.

isSatisfyRemoteProcCallBodyCondition0(*bl*: *BINDINGLIST0*, *body*: <remote procedure call body>):
BOOLEAN =$_{def}$
let *rpd = getEntityDefinition0*(*body*.**s-**<remote procedure<identifier>,**remote procedure**) **in**
let *fpsl= rpd.formalParameterSortList0* **in**
let *apl = body.actualParameterList0* **in**
fpsl.length=apl.length ∧
(∀*i*∈1..*fpsl.length*: *isSortCompatible0*(*getStaticSort0*(*apl*[*i*], *bl*), *fpsl*[*i*])∧
isSatisfyExpressionCondition0(*bl*, *apl*[*i*]))
endlet
endlet
endlet

The function *operationResultSort0* gets the result sort of an <operation signature>.

operationResultSort0(*os*:<operation signatures>):<sort>=$_{def}$
if *os*∈*PREDEFINEDOPERATION0* **then** *os*. *getPredefinedOpResult0*
else *os*.**s-**<result>.**s-**<sort>
endif

The function *isSatisfyRangeCheckCondition0* determines if the binding violates any static sort constraints in the <range check expression>.

isSatisfyRangeCheckCondition0(*bl*: *BINDINGLIST0*, *rce*: <range check expression>): *BOOLEAN*=$_{\text{def}}$
  if *rce*.**s-implicit**∈<sort> **then**
    isSatisfyExpressionCondition0(*bl*, *rce*.**s-**<expression>)∧
    isSameSort0(*getStaticSort0*(*rce*.**s-**<expression>, *bl*), *rce*.**s-implicit**∈<sort>)
  else
    isSatisfyExpressionCondition0(*bl*, *rce*.**s-**<expression>)∧
    isSameSort0(*getStaticSort0*(*rce*.**s-**<operand2>, *bl*), *rce*.**s-implicit**.**s-**<sort<identifier>)
  endif

The function *isSatisfyEqualityExpCondition0* determines if the binding violates any static sort constraints in the <equality expression>.

isSatisfyEqualityExpCondition0(*bl*: *BINDINGLIST0*, *eq*: <equality expression>): *BOOLEAN*=$_{\text{def}}$
  isSatisfyExpressionCondition0(*bl*, *eq*.**s-**<expression>)∧
  isSatisfyExpressionCondition0(*bl*, *eq*.**s2-**<expression>)∧
  (*isSortCompatible0*(*getStaticSort0*(*eq*.**s-**<expression>, *bl*), *getStaticSort0*(*eq*.**s2-**<expression>, *bl*))∨
    isSortCompatible0(*getStaticSort0*(*eq*.**s2-**<expression>, *bl*),
      getStaticSort0(*eq*.**s-**<expression>, *bl*)))

The function *isSatisfyPrimaryCondition0* determines if the binding violates any static sort constraints in the <primary>.

isSatisfyPrimaryCondition0(*bl*: *BINDINGLIST0*, *pr*: <primary>):*BOOLEAN*=$_{\text{def}}$
  case *pr* **of**
  |<operator application>=>*isSatisfyOpAppCondition0*(*bl*, *pr*)
  |<method application>=>*isSatisfyMethodAppCondition0*(*bl*, *pr*)
  |<expression>=>*isSatisfyExpressionCondition0*(*bl*, *pr*)
  |<conditional expression>=>
    isSatisfyExpressionCondition0(*bl*, *pr*.**s-**<expression>)∧
    isSatisfyExpressionCondition0(*bl*, *pr*.**s-**<consequence expression>)∧
    isSatisfyExpressionCondition0(*bl*, *pr*.**s-**<alternative expression>)
  |<extended primary>=>*isSatisfyExtendedPrimaryCond0*(*bl*, *pr*)
  otherwise *true*
  endcase

The function *isSatisfyExtendedPrimaryCond0* determines if the binding violates any static sort constraints in the <extended primary>.

isSatisfyExtendedPrimaryCond0(*bl*: *BINDINGLIST0*, *epr*: <extended primary>):*BOOLEAN*=$_{\text{def}}$
  case *epr* **of**
  |<indexed primary>=>
    isSatisfyPrimaryCondition0(*epr*.**s-**<primary>, *bl*)∧
    (∀*i*∈1..*epr*.**s-**<actual parameter>-**seq**.*length*:
      isSatisfyExpressionCondition0(*bl*,*epr*.**s-**<actual parameter>-**seq**[*i*]))∧
    isSortCompatible0(*getStaticSort0*(*epr*.**s-**<actual parameter>-**seq**[1], *bl*),
      getIndexSort0(*getPrimarySort0*(*epr*.**s-**<primary>, *bl*)))
  |<field primary>=>
    (*epr*.**s-**<primary> ≠ *undefined*)⇒
    (*isSatisfyPrimaryCondition0*(*epr*.**s-**<primary>, *bl*)∧
      getFieldSort0(*getPrimarySort0*(*epr*.**s-**<primary>, *bl*), *epr*.**s-**<field name>) ≠ *undefined*)
  |<composite primary>=>
    let *sl* = <*getStaticSort0*(*para, bl*)| *para* **in** *epr*.**s-**<actual parameter>-**seq**> **in**
      epr.**s-**<actual parameter>-**seq**≠*empty*⇒
      (*getCompositeSort0*(*sl*)≠ *undefined* ∧
      (∀*i*∈1..*epr*.**s-**<actual parameter>-**seq**.*length*: *epr*.**s-**<actual parameter>-**seq**[*i*]= *undefined*∨
        isSatisfyExpressionCondition0(*bl*,*epr*.**s-**<actual parameter>-**seq**[*i*])))
    endlet
  endcase

The function *getCompositeSort0* gets the sort that refers to a structure data type whose field sort list is the same as the specified parameters.

```
getCompositeSort0(sl:[<sort>]*):<sort>= def
    let def = take({ d∈<data type definition>∪<syntype definition>:
                    ∃cons∈<structure definition>:
                    let fsl = cons.fieldSortList0 in
                        fsl.length=sl.length∧
                        (∀i∈1..fsl.length: sl[i]=undefined∨isSortCompatible0(sl[i], fsl[i]))
                    endlet}) in
        def.derivedDataType0.identifier0
    endlet
```

The function *getPrimarySort0* gets the static sort of a <primary>.

```
getPrimarySort0(pr: <primary>, bl: BINDINGLIST0): <sort>= def
    case pr of
    |<operation application>=>getOpAppSort0(pr, bl)
    |<expression>=>getStaticSort0( pr, bl)
    |<conditional expression>=>getStaticSort0( pr.s-<consequence expression>, bl)
    |<literal>=>
        let ls = getDefinitionInBindingList0 (pr, bl) in
        ls.surroundingScopeUnit0.identifier0
    |<spelling term>=>
        <identifier>(<qualifier>( < <path item>( package, <name>("Predefined") ) > ),
            <name>("Charstring"))
    |<indexed primary>=>getItemSort0(getPrimarySort0(epr.s-<primary>, bl))
    |<field primary>=>getFieldSort0(getPrimarySort0(epr.s-<primary>, bl), epr.s-<field name>)
    |<composite primary>=>
        let sl = <getStaticSort0(para, bl)| para in epr.s-<actual parameter>-seq> in
            getCompositeSort0(sl)
        endlet
    |<variable access>=>getVarAccessSort0(pr)
    |<imperative expression>=>getImperativeExpSort0(pr)
    |<synonym>=>getSynonymSort0(pr)
    otherwise true
    endcase
```

The function *getVarAccessSort0* gets the static sort of a <variable access>.

```
getVarAccessSort0(va: <variable access>): <sort>= def
    if va∈ <identifier> then
        getEntityDefinition0(va, variable).s-<sort>
    else
        let od = parentAS0ofKind(va, <operation definition>) in
            od.operationFormalparameterList0[1].parentAS1.s-<sort>
        endlet
    endif
```

The function *getImperativeExpSort0* gets the static sort of an <imperative expression>.

```
getImperativeExpSort0(ie: <imperative expression>): <sort>= def
    case ie of
    |<now expression>=> <identifier>(<qualifier>(<name>("Predefined")), "Time")
    |<pid expression>=>
        if ie ≠ self then
            <identifier>(<qualifier>( < <path item>( package, <name>("Predefined") ) > ),
                <name>("Pid"))
        else
            let def = parentAS0ofKind(ie, <agent definition>∪<agent type definition>) in
            if def = undefined then
                <identifier>(<qualifier>( < <path item>( package, <name>("Predefined") ) > ),
                    <name>("Pid"))
```

```
            else def.identifier0
            endif   endlet
        endif
    |<any expression>=>ie.s-<sort>
    |<state expression>=>
        <identifier>(<qualifier>( < <path item>( package, <name>("Predefined") ) > ),
            <name>("Charstring"))
endcase
```

The function *getSynonymSort0* gets the static sort of a <synonym>.

```
getSynonymSort0(s: <synonym>): <sort>= def
    let sd = getEntityDefinition0(pr, synonym) in
    if sd.s-<sort> ≠ undefined then sd.s-<sort>
    else take(sd.s-<constant expression>.staticSortSet0)
    endif endlet
```

The function *isSatisfyOpAppCondition0* determines if the binding violates any static sort constraints in the <operator application>.

```
isSatisfyOpAppCondition0(bl:BINDINGLIST0, oa:<operation application>): BOOLEAN=def
    let opDef = getDefinitionInBindingList0(oa.s-<operation identifier>.s-<name>, bl) in
    let fpl = opDef.operationParameterSortList0 in
    let apl =oa.actualParameterList0 in
    fpl.length = apl.length ∧
    (∀i∈1..fpl.length: isSortCompatible0(getStaticSort0(apl[i], bl), fpl[i])∧
        isSatisfyExpressionCondition0(bl, apl[i])
    endlet
```

The function *getOpAppSort0* gets the static sort of a <synonym>.

```
getOpAppSort0(oa:<operation application>, bl: BINDINGLIST0): <sort>= def
    getDefinitionInBindingList0(oa.s-<operation identifier>.s-<name>, bl).operationResultSort0
```

The function *isSatisfyMethodAppCondition0* determines if the binding violates any static sort constraints in the <method application>.

```
isSatisfyMethodAppCondition0(bl:BINDINGLIST0, ma: <method application>): BOOLEAN=def
    let opDef = getDefinitionInBindingList0(ma.s-<operation identifier>.s-<name>, bl) in
    let fpl = opDef.operationParameterSortList0 in
    let apl =ma.actualParameterList0 in
    fpl.length = apl.length ∧isSatisfyPrimaryCondition0(bl, ma.s-<primary>)∧
    (∀i∈1..fpl.length: isSortCompatible0(getStaticSort0(apl[i], bl), fpl[i])∧
        isSatisfyExpressionCondition0(bl, apl[i])
    endlet
```

The function *operationParameterSortList0* gets the operation formal parameter sort list.

```
operationParameterSortList0(os:<operation signature>):<sort>*=def
    if os. getPredefinedOpParas0 ≠ undefined then
        os.getPredefinedOpParas0
    else
            <paras.s-<formal parameter>.s-<sort> | paras in os.operationSignatureParameterList0>
    endif
```

The function *getDefinitionInBindingList0* gets the corresponding entity definition for a name in a binding list.

```
getDefinitionInBindingList0 (n: <name>, bl: BINDINGLIST0): ENTITYDEFINITION0=def
    take ({d∈ENTITYDEFINITION0: ∃i∈1..bl.length: bl[i].s-<name>=n∧ bl[i].s-ENTITYDEFINITION0=d})
```

The function *possibleDefinitionSet0* gets the set of possible entity definition for a name.

```
possibleDefinitionSet0(n: <name>):ENTITYDEFINITION0-set =def
    {d∈ENTITYDEFINITION0: ((d.entityName0=n)∨(isRenamedBy0(d.entityName0, n)))∧
        (   (d.entityKind0=n.idKind0∧isVisibleIn0(d, n.surroundingScopeUnit0))∨
            (d∈PREDEFINEDLITERAL0∧ n.idKind0=literal)∨
            (d∈PREDEFINEDOPERATION0∧ n.idKind0∈{operator, method}))}
```

The function *isRenamedBy0* determines if a name is renamed by another one.

```
isRenamedBy0(n1, n2: <name>):BOOLEAN=def
    (∃rp∈<rename pair>:(rp.s-<operation name> = n2∧ rp.s2-<operation name>=n1)∨
                       (rp.s-<literal name> = n2∧ rp.s2-<literal name>=n1))∨
    (∃n3∈<name>: isRenamedBy0(n1, n3)∧ isRenamedBy0(n3, n2))
```

The function *actualParameterList0* gets the actual parameter list for a <create expression>, a <procedure call body>, a <remote procedure call body>, a <value returning procedure call> or an <operation application> or a <method application>.

```
actualParameterList0(d:
<create expression>∪<procedure call body>∪<remote procedure call body>
∪<value returning procedure call>∪<operation application>): <expression>*=def
    d. s-<actual parameter>-seq
```

The function *getCreatedAgentDefinition0* gets the <agent definition> or <agent type definition> that the <create expression> involves.

```
getCreatedAgentDefinition0 (ce: <create expression>): <agent definition>∪<agent type definition>=def
    let id= ce.s-implicit in
    if id∈<identifier> then getEntityDefinition0(id, id.idKind0)
    elseif id.surroundingScopeUnit0∈<agent definition>∪<agent type definition> then
            id.surroundingScopeUnit0
    else undefined
    endif
```

The function *getStaticSort0* gets the static sort of an expression.

```
getStaticSort0(exp: <expression>, bl:BINDINGLIST0): <sort>=def
    case exp of
    |<create expression>=>getCreateExpSort0(exp)
    |<value returning procedure call>=>
        if exp∈<procedure call body> then getProcCallBodySort0(exp)
        else getRemoteProcCallBodySort0(exp, bl)
        endif
    |<range check expression> => <identifier>(
        <qualifier>( < <path item>( package, <name>("Predefined") ) > ), <name>("Boolean"))
    |<equality expression>=> <identifier>(
        <qualifier>( < <path item>( package, <name>("Predefined") ) > ), <name>("Boolean"))
    |<binary expression>=> getDefinitionInBindingList0(exp.s-implicit, bl).operationResultSort0
    |<operand5>=>
        if exp.s-implicit = undefined then
            getPrimarySort0(exp, bl)
        else
            let opDef = getDefinitionInBindingList0(exp.s-implicit, bl) in
                opDef.operationResultSort0
            endlet
        endif
    endcase
```

The function *mismatchNumber0* counts the number of mismatches that the static sort of an <expression> is not the same as the static sort of the <variable> or the static sort of an actual parameter is not the same as the sort of the corresponding formal parameter.

```
mismatchNumber0(bl:BINDINGLIST0, c:CONTEXT0): NAT=def
```

```
    case c of
    |<assignment>=>mismatchNumberOfAssignment0(bl, c)
    |<decision>=>mismatchNumberOfDecision0(bl, c)
    |<expression>=>mismatchNumberOfExpression0(bl, c)
    endcase

mismatchNumberOfAssignment0(bl:BINDINGLIST0, ass: <assignment>): NAT=def
    let varSort= getVariableSort0(ass.s-<variable>) in
    let expSort= getStaticSort0(ass.s-<expression>, bl) in
    case ass.s-<variable> of
    | <identifier>∪<field variable>=>
        if ¬isSameSort0(varSort, expSort)
        then 1+mismatchNumberOfExpression0(bl, ass.s-<expression>)
        else mismatchNumberOfExpression0(bl, ass.s-<expression>)
        endif
    |<indexed variable>=>
        if ¬isSameSort0(varSort, expSort) then
            1 + mismatchNumberOfExpression0(bl, ass.s-<expression>)+
                mismatchNumberInIndexVariable0(bl, ass.s-<variable>)
        else
            mismatchNumberOfExpression0(bl, ass.s-<expression>)+
            mismatchNumberInIndexVariable0(bl, ass.s-<variable>)
        endif
    endcase

mismatchNumberInIndexVariable0(bl:BINDINGLIST0, var:<indexed variable>): NAT=def
    let acp=var.s-<actual parameter>-seq in
        if var.s-<variable>∉<indexed variable> then
            if ¬isSameSort0(getStaticSort0(acp[1],bl), getIndexSort0(getVariableSort0(var))) then 1
            else 0
            endif
        elseif ¬isSameSort0(getStaticSort0(acp[1],bl), getIndexSort0(getVariableSort0(var))) then
            1 + mismatchNumberInIndexVariable0(bl, var.s-<variable>)
        else mismatchNumberInIndexVariable0(bl, var.s-<variable>)
        endif
    endlet

mismatchNumberOfDecision0(bl:BINDINGLIST0, d: <decision>): NAT=def
    let q=d.s-<question>, rcl=d.rangeConditionList0 in
        mismatchNumberOfExpression0(bl, q) + mismatchNumberOfRangeConditionList0(bl, rcl))
    endlet

mismatchNumberOfRangeConditionList0(bl:BINDINGLIST0, rcl: <range condition>*): NAT=def
    if rcl.= empty then empty
    else
        mismatchNumberOfRangeCond0(bl, rcl.head) +
            mismatchNumberOfRangeConditionList0 (bl, rcl.tail])
    endif

mismatchNumberOfRangeCond0(bl:BINDINGLIST0, rl: <range>*): NAT=def
    if rl = empty then empty
    else
        mismatchNumberOfRange0(bl, rl.head)+ mismatchNumberOfRangeCond0 (bl, rl.tail])
    endif

mismatchNumberOfRange0(bl:BINDINGLIST0, range: <range>): NAT=def
    case range of
    |<closed range>=>
        mismatchNumberOfExpression0(bl,range.s-<constant>) +
        mismatchNumberOfExpression0(bl,range.s2-<constant> )
    |<open range>=>
        if range∈<constant> then mismatchNumberOfExpression0(bl,range)
```

```
        else
            mismatchNumberOfExpression0(bl,range.s-<constant>)+
            mismatchNumberOfExpression0(bl,range.s2-<constant> )
        endif
    endcase

mismatchNumberOfExpression0(bl:BINDINGLIST0, exp: <expression>): NAT=def
    case exp of
    |<create expression>=>mismatchNumberOfCreateExp0(bl, exp)
    |<value returning procedure call>=>
        if exp∈<procedure call body> then
            mismatchNumberOfProcedureCallBody0(bl, call)
        else
            mismatchNumberOfRemoteProcCallBody0(bl, call)
        endif
    |<range check expression> => mismatchNumberOfExpression0(bl,exp.s-<expression>)
    |<equality expression>=>
        mismatchNumberOfExpression0(bl,exp.s-<expression>)+
        mismatchNumberOfExpression0(bl,exp.s2-<expression>)
    |<binary expression>=>
        let opDef = getDefinitionInBindingList0(exp.s-implicit, bl) in
        let fpsl = opDef.operationParameterSortList0 in
            mismatchNumberOfParas0(bl, fpsl, exp.s-<expression>⌒ exp.s2-<expression>)+
            mismatchNumberOfExpression0(bl, exp.s-<expression>)+
            mismatchNumberOfExpression0(bl, exp.s2-<expression>)
        endlet
    |<operand5>=>
        if exp.s-implicit = undefined then
            mismatchNumberOfPrimary0(bl, pr)
        else
            let opDef = getDefinitionInBindingList0(exp.s-implicit, bl) in
            let fpsl = opDef.operationParameterSortList0 in
            if isSameSort0(getStaticSort0(pr, bl), fpsl[1]) then mismatchNumberOfPrimary0(bl, pr)
            else mismatchNumberOfPrimary0(bl, pr)+1
            endif endlet
        endif
    endcase

mismatchNumberOfCreateExp0(bl: BINDINGLIST0, ce: <create expression>):NAT=def
    let def = ce.getCreatedAgentDefinition0 in
        if def =undefined then 0
        else
            mismatchNumberOfParas0(bl, def.formalParameterSortList0, ce.actualParameterList0) +
                mismatchNumberOfActualParas0(bl, ce.actualParameterList0)
        endif
    endlet

mismatchNumberOfProcedureCallBody0(bl: BINDINGLIST0, body: <procedure call body>): NAT=def
    case body.s-implicit of
    | id <identifier> =>
        let fpsl=getEntityDefinition0(id, procedure).formalParameterSortList0 in
            mismatchNumberOfParas0(bl, fpsl, apl)
        endlet
    | te=<type expression>=>
        let fpsl=id.baseType0.formalParameterSortList0 in
        let apl = body.actualParameterList0 in
        endlet
        let apl = body.actualParameterList0 in
        endlet
            mismatchNumberOfParas0(bl, fpsl, apl) + mismatchNumberOfActualParas0(bl, apl)
        endlet
    endcase
```

mismatchNumberOfRemoteProcCallBody0(*bl*: *BINDINGLIST0*, *body*: <remote procedure call body>):
NAT=$_{def}$
let *rpd* = *getEntityDefinition0*(*body*.**s-** <identifier>,**remote procedure**) **in**
let *fpsl*= *rpd.formalParameterSortList0* **in**
let *apl* = *body.actualParameterList0* **in**
mismatchNumberOfParas0(*bl*, *fpsl*, *apl*) + *mismatchNumberOfActualParas0*(*bl*, *apl*)
endlet
endlet
endlet

mismatchNumberOfPrimary0(*bl*: *BINDINGLIST0*, *pr*: <primary>):*NAT*=$_{def}$
case *pr* **of**
|<operator application>=>*mismatchNumberOfOpApp0*(*bl*, *pr*)
|<method application>=>*mismatchNumberOfMethodApp0*(*bl*, *pr*)
|<expression>=>*mismatchNumberOfExpression0*(*bl*, *pr*)
|<conditional expression>=>
mismatchNumberOfExpression0(*bl*, *pr*.**s-**<expression>) +
mismatchNumberOfExpression0(*bl*, *pr*.**s-**<consequence expression>) +
mismatchNumberOfExpression0(*bl*, *pr*.**s-**<alternative expression>)
otherwise 0
endcase

mismatchNumberOfOpApp0(*bl*:*BINDINGLIST0*, *oa*: <operator application>): *NAT*=$_{def}$
let *opDef* = *getDefinitionInBindingList0*(*oa*.**s-**<operation identifier>.**s-**<name>, *bl*) **in**
let *fpsl* = *opDef.operationParameterSortList0* **in**
let *apl* =*oa.actualParameterList0* **in**
mismatchNumberOfParas0(*bl*, *fpsl*, *apl*)+*mismatchNumberOfActualParas0*(*bl*, *apl*)
endlet
endlet
endlet

mismatchNumberOfMethodApp0(*bl*:*BINDINGLIST0*, *ma*: <method application>):*NAT*=$_{def}$
let *opDef* = *getDefinitionInBindingList0*(*ma*.**s-**<operation identifier>.**s-**<name>, *bl*) **in**
let *fpsl* = *opDef.operationParameterSortList0* **in**
let *apl* =*ma.actualParameterList0* **in**
mismatchNumberOfParas0(*bl*, *fpsl*, *apl*)+*mismatchNumberOfActualParas0*(*bl*, *apl*)+
mismatchNumberOfPrimary0(*bl*, *ma*.**s-**<primary>)
endlet
endlet
endlet

mismatchNumberOfParas0(*bl*: *BINDINGLIST0*, *fpsl*: <sort>*, *apl*: <expression>*):*NAT*=$_{def}$
if *fpsl* = *empty* **then** 0
elseif *fpsl.head=undefined*∨*isSameSort0*(*fpsl.head*, *apl.head*)
then *mismatchNumberOfParas0*(*bl*, *fpsl.tail*, *apl.tail*)
else *mismatchNumberOfParas0*(*bl*, *fpsl.tail*, *apl.tail*)+1
endif

mismatchNumberOfActualParas0(*bl*: *BINDINGLIST0*, *apl*: <expression>*): *NAT*=$_{def}$
if *apl* = *empty* **then** 0
else *mismatchNumberOfExpression0*(*bl*, *apl.head*)+*mismatchNumberOfActualParas0*(*bl*, *apl.tail*)
endif

formalParameterSortList0(*d*:
<agent definition> ∪ <agent type definition> ∪ <composite state> ∪
<composite state type definition> ∪ <procedure definition> ∪
<remote procedure definition>):<sort>*= def
case *d* **of**
| <agent definition> ∪ <agent type definition>∪
<composite state> ∪ <composite state type definition>=>
<*fp.parentAS1*.**s-**<sort> | *fp* **in** *d.agentFormalParameterList0*>
|<procedure definition>=>

<*fp.parentAS1*.**s-**<sort> | *fp* **in** *d.procedureFormalParameterList0*>
|<remote procedure definition>=>
<*fp*.**s-**<sort> | *fp* **in** *d*.**s-**<procedure signature>.**s-**<formal parameter>>
endcase

The function *staticSortSet0* gets the possible static sort set for an <expression>.

staticSortSet0(*exp*: <expression>):<sort>-**set** =$_{def}$
{*getStaticSort0*(*exp, bl*): *bl*∈*getBindingListSet0*(*exp.contextOfExp0*)}

The function *contextOfExp0* finds the context that the <expression> is in.

contextOfExp0(*exp*: <expression>):*CONTEXT0*=$_{def}$
if (∃*ass*∈<assignment>: *isAncestorAS0*(*ass*, *exp*)) **then**
parentAS0ofKind(*exp*, <assignment>)
elseif (∃*ass*∈<decision>: *isAncestorAS1*(*ass*, *exp*)) **then**
parentAS0ofKind(*exp*, <decision>)
elseif (∃*exp1*∈<expression>: *isAncestorAS0*(*exp1*, *exp*)) **then**
parentAS0ofKind(*exp*, <expression>).*contextOfExp0*
endif

The function *getPredefinedOpParas0* gets the sort list of the formal parameters of a predefined operation. This function is not formally defined in this Recommendation (ITU-T Z.100).

getPredefinedOpParas0: *PREDEFINEDOPERATION0* → <sort>*

The function *getPredefinedOpResult0* gets the result sort of a predefined operation. This function is not formally defined in this Recommendation (ITU-T Z.100).

getPredefinedOpResult0: *PREDEFINEDOPERATION0* → <sort>

F2.2.2.3 Path item

Abstract syntax

Path-item	=	*Package-qualifier*
	|	*Agent-type-qualifier*
	|	*Agent-qualifier*
	|	*State-type-qualifier*
	|	*State-qualifier*
	|	*Data-type-qualifier*
	|	*Procedure-qualifier*
	|	*Interface-qualifier*
Package-qualifier	::	*Package-name*
Agent-type-qualifier	::	*Agent-type-name*
Agent-qualifier	::	*Agent-name*
State-type-qualifier	::	*State-type-name*
State-qualifier	::	*State-name*
Data-type-qualifier	::	*Data-type-name*
Procedure-qualifier	::	*Procedure-name*
Interface-qualifier	::	*Interface-name*

Concrete syntax

<path item>	::	<scope unit kind> <name>
<scope unit kind>	=	
		package
	|	**system type**
	|	**system**
	|	**block**

| **block type**
| **process**
| **process type**
| **state**
| **state type**
| **procedure**
| **signal**
| **type**
| **operator**
| **method**
| **interface**

Mapping to abstract syntax

| <path item>(**package**,*n*) => **mk-***Package-qualifier*(*n*)
| <path item>(**system**,*n*) => **mk-***Agent-qualifier*(*n*)
| <path item>(**system type**,*n*) => **mk-***Agent-type-qualifier*(*n*)
| <path item>(**block**,*n*) => **mk-***Agent-qualifier*(*n*)
| <path item>(**block type**,*n*) => **mk-***Agent-type-qualifier*(*n*)
| <path item>(**process**,*n*) => **mk-***Agent-qualifier*(*n*)
| <path item>(**process type**,*n*) => **mk-***Agent-type-qualifier*(*n*)
| <path item>(**state**,*n*) => **mk-***State-qualifier*(*n*)
| <path item>(**statetype**,*n*) => **mk-***State-type-qualifier*(*n*)
| <path item>(**procedure**,*n*) => **mk-***Procedure-qualifier*(*n*)
| <path item>(**operator**,*n*) => **mk-***Procedure-qualifier*(*n*)
| <path item>(**method**,*n*) => **mk-***Procedure-qualifier*(*n*)
| <path item>(**type**,*n*) => **mk-***Data-type-qualifier*(*n*)
| <path item>(**interface**,*n*) => **mk-***Interface-qualifier*(*n*)

F2.2.3 Informal text

Abstract syntax

Informal-text :: ...

Concrete syntax

<informal text> :: <character string>

Mapping to abstract syntax

The mapping for informal text is described in clause F2.2.2.1.

F2.2.4 General framework

F2.2.4.1 SDL-2010 specification

Abstract syntax

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

Concrete syntax

<sdl specification> ::
{ <textual system specification> | <package definition> } <referenced definition>*

<textual system specification> =
<agent definition>
| <package use clause>* <textual typebased agent definition>

Transformations

<sdl specification>(*sys*, *p*, *r*)
provided *sys* ∈ (<process definition> ∪ <textual typebased process definition> ∪
<block definition> ∪ <textual typebased block definition>)
=1=> <sdl specification>(

<system definition>(*empty*,
<system heading>(*sys.name0*, <agent additional heading>(*undefined*, *empty*)),
<agent structure>(*undefined*, < *sys* >, <agent body>(*undefined*, *empty*))),
p,*r*)

A <system specification> being a <process definition> or a <textual typebased process definition> is derived syntax for a <block<system definition> having the same name as the process, containing implicit channels and containing the <process definition> or <textual typebased process definition> as the only definition.

A <system specification> being a <block definition> or a <textual typebased block definition> is derived syntax for a <system definition> having the same name as the block, containing implicit channels and containing the <block definition> or <textual typebased block definition> as the only definition.

Mapping to abstract syntax

| <sdl specification>(*s*, *packages*, *) =>
mk-*Sdl-specification*(*Mapping*(*s*), *Mapping*(*packages*))

F2.2.4.2 Package

Abstract syntax

Package-definition :: *Package-name*
Package-definition-**set**
Data-type-definition-**set**
Syntype-definition-**set**
Signal-definition-**set**
Agent-type-definition-**set**
Composite-state-type-definition-**set**
Procedure-definition-**set**

Concrete syntax

<package definition> :: <package use clause>* <package heading> <entity in package>*

<package heading> :: <qualifier> <package<name> <package public>

<entity in package> =
<agent type definition>
| <agent type reference>
| <package definition>
| <package reference>
| <signal definition list>
| <signal list definition>
| <remote variable definition>
| <data definition>
| <procedure definition>
| <procedure reference>
| <remote procedure definition>
| <composite state type definition>
| <composite state type reference>
| <select definition>

<package use clause> :: <package<identifier> <definition selection>*

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

<selected entity kind> =
system type
| **block type**
| **process type**
| **package**
| **signal**
| **procedure**

| **remote procedure**
| **type**
| **signallist**
| **state type**
| **synonym**
| **remote**
| **interface**

<package public> = <definition selection>*

Conditions on concrete syntax

∀*id*∈ <identifier>:
(*id*.*parentAS0*∈<package use clause>)⇒*getEntityDefinition0*(*id*, **package**) ≠ *undefined*

For each <package<identifier> mentioned in a <package use clause>, there must exist a corresponding visible <package>.

∀*pd*∈ <package definition>:
pd. *parentAS0*∈<sdl specification> ⇒ *pd.qualifier0* = *undefined*

If the package is part of <sdl specification>, there must not be a <qualifier> in <package identifier>.

∀*ds1*,*ds2*∈<definition selection>:
(*ds1*. *parentAS0* = *ds2*. *parentAS0*∧ *ds1* ≠ *ds2*)⇒
(*ds1*.**s-**<selected entity kind> ≠ *ds2*. **s-**<selected entity kind> ∨ *ds1*.**s-**<name> ≠ *ds2*.**s-**<name>)

Any pair of (<selected entity kind>, <name>) must be distinct within a <definition selection list>.

∀*ds*∈<definition selection>:
ds.**s-**<selected entity kind> = *undefined*∧ *d*. *parentAS0*∈<package public> ⇒
(∃! *d*∈ *ENTITYDEFINITION0*∪ *REFERENCE0* :
d.*surroundingScopeUnit0* = *ds*. *parentAS0*. *parentAS0* ∧ *d*.*entityName0*= *ds*.**s-**<name>)

For a <definition selection> in a <package public>, the <selected entity kind> may be omitted only if there is no other name having its defining occurrence directly in the <package>.

∀ *uc* ∈ <package use clause>:∀*ds*∈<definition selection>:
let *pd* = *uc*.*usedPackage0* **in**
ds. *parentAS0* = *uc* ∧ *ds*.**s-**<selected entity kind> = *undefined*⇒
((∃!*ds1*∈<definition selection>:
ds1.*surroundingScopeUnit0*= *pd*∧ *ds*.**s-**<name>= *ds1*.**s-**<name>) ∨
((*pd*. **s-**<package heading>.**s-**<package public> = *undefined*)∧
(∃!*d*∈ *ENTITYDEFINITION0*∪ *REFERENCE0* :
d.*surroundingScopeUnit0*= *pd* ∧ *d*. *entityName0*= *ds*.**s-**<name>)))
endlet

For a <definition selection> in a <package use clause>, <selected entity kind> may be omitted if and only if either exactly one entity of that name is mentioned in any <definition selection list> for the package or the package has no <definition selection list> and directly contains a unique definition of that name.

∀*ds*∈<definition selection>:
ds.**s-**<selected entity kind> ≠ *undefined*∧ *d*. *parentAS0* ∈<package public> ⇒
(∃ *d*∈*ENTITYDEFINITION0*∪ *REFERENCE0* :
d.*surroundingScopeUnit0* = *ds*.*surroundingScopeUnit0*∧ *d*. *entityName0* = *ds*.**s-**<name> ∧
d.*entityKind0* = *ds*.**s-**<selected entity kind>)

For each pair of (<selected entity kind>, <name>) in <package public>, there must exist an entity definition having the same entity kind and the same name in the package.

Transformations

let *usePredef* = <package use clause>(
<identifier>(<qualifier>(< <path item>(**package**, <name>("Predefined")) >), *undefined*) **in**

<package definition>(*uses*, *heading*, *entities*) **provided** *uses.head* ≠ *usePredef* =6=>
    <package definition>(< *usePredef* > ⌒ *uses*, *heading*, *entities*)
endlet

let *usePredef* = <package use clause>(
    <identifier>(<qualifier>(< <path item>(**package**, <name>("Predefined")) >), *undefined*) **in**
<system definition>(*uses*, *heading*, *struct*) **provided** *uses.head* ≠ *usePredef* =6=>
    <system definition>(< *usePredef* > ⌒ *uses*, *heading*, *struct*)
endlet

A <system definition> and every <package definition> have an implicit <package use clause>:

use Predefined;

where Predefined denotes a package containing the predefined data as defined in clause 14 of [ITU-T Z.104].

< <package use clause>(*id*, *sel1*) > ⌒ *something* ⌒ < <package use clause>(*id*, *sel2*) >
=8=>
    (**let** *newSel* =
        if *sel1* = *undefined* **then** *empty*
        elseif *sel2* = *undefined* **then** *empty*
        else *sel1* ⌒ < *s* **in** *sel2*: (*s* ∉ *sel1.toSet*) >
        endif
    in
        < <package use clause>(*id*, *newSel*) > ⌒ *something*
    endlet)

If a package is mentioned in several <package use clause>s of a <package definition>, this corresponds to one <package use clause> that selects the union of the definitions selected in the <package use clause>s.

Mapping to abstract syntax

<package definition>(*, <package heading>(*, *n*, *), *entities*)
=> **mk-***Package-definition*(*n*,
        { *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Package-definition*) },
        { *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Data-type-definition*) },
        { *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Syntype-definition*) },
        { *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Signal-definition*) },
        { *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Agent-type-definition*) },
        { *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Composite-state-type-definition*) },
        { *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Procedure-definition*) })

Auxiliary functions

The function *usedPackageDefinitionList0* gets the used package definition list of a scope unit.

usedPackageDefinitionList0(*su*: *SCOPEUNIT0*): <package definition> *=$_{def}$
    < *u.usedPackage0* | *u* **in** *su*.**s-**<package use clause> >

The function *usedPackage0* gets the package definition for a <package use clause>.

usedPackage0 (*uc*: <package use clause>): <package definition> =$_{def}$
    getEntityDefinition0 (*uc*.**s-**<identifier>, **package**)

F2.2.4.3 Referenced definition

Concrete syntax

<referenced definition> =

<definition>

<definition> =
| <package definition>
| <agent definition>
| <agent type definition>
| <composite state>
| <composite state type definition>
| <procedure definition>
| <operation definition>

<package reference> :: <package<identifier>

<procedure reference> :: <procedure reference heading>

<procedure reference heading> =
<type preamble> [<exported>] [<qualifier>] <procedure<name> [<formal context parameters>]

<block reference> :: <block<name> <number of instances>

<process reference> :: <process<name> <number of instances>

<composite state reference> :: <composite state name>

<agent type reference> =
<system type reference>
| <block type reference>
| <process type reference>

<system type reference> :: <system type<identifier> [<formal context parameters>]

<block type reference> :: <type preamble> <block type<identifier> [<formal context parameters>]

<process type reference> :: <type preamble> <process type<identifier> [<formal context parameters>]

<composite state type reference> :: <type preamble> <composite state type<identifier>

Conditions on concrete syntax

∀*refDef*∈<referenced definition>: *refDef.referencedBy0*≠ *undefined*

For each <referenced definition>, there must be a reference in the associated <package> or <system specification>.

∀*ref*∈ *REFERENCE0* : *ref.referencedDefinition0* ≠ *undefined*

For each reference there must exist a <referenced definition> with the same entity kind as the reference, and whose <qualifier>, if present, denotes a path, from a scope unit enclosing the reference, to the reference.

∀*ref*∈ *REFERENCE0* : ∀*refDef*∈<referenced definition>:
ref.**s-**<formal context parameters> ≠ *undefined* ∧ *refDef* = *ref.referencedDefinition0* ⇒
refDef.**s-**<formal context parameters> ≠ *undefined* ∧
isFormalContextParametersMatched0(*ref*,*refDef*)

If there is a <formal context parameters> item in the type reference, this shall be the same as the <formal context parameters> of the <referenced definition>.

∀*ref1*, *ref2* ∈<referenced definition>:
ref1.entityName0= *ref2.entityName0*∧ *ref1.entityKind0*= *ref2.entityKind0* ∧ *ref1* ≠ *ref2*⇒
ref1.qualifier0≠ *undefined*∧ *ref2.qualifier0*≠ *undefined*∧
¬*isPathItemMatched0* (*ref1.qualifier0*.**s-**<path item>**-seq**, *ref2.qualifier0*.**s-**<path item>**-seq**) ∧
¬*isPathItemMatched0* (*ref2.qualifier0*.**s-**<path item>**-seq**, *ref1.qualifier0*.**s-**<path item>**-seq**)

If two <referenced definition>s of the same entity kind have the same <name>, the <qualifier> of one must not constitute the leftmost part of the other <qualifier>, and neither <qualifier> can be omitted.

∀*rd*∈ <referenced definition>:
rd∈<package definition>⇒ *rd*. *qualifier0* ≠ *undefined*

The <qualifier> must be present if the <referenced definition> is a <package definition>.

$\forall$ *def*$\in$*ENTITYDEFINITION0*: *def*$\notin$<referenced definition>$\Rightarrow$ *def.qualifier0= undefined*

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

Transformations

p= <package reference>(*) =4=> *adaptDefinition*(*p.referencedDefinition0, undefined*)

p= <procedure reference>(*,*) =4=> *adaptDefinition*(*p.referencedDefinition0, undefined*)

b= <block reference>(*, *inst*) =4=> *adaptDefinition*(*b.referencedDefinition0, inst*)

p= <process reference>(*, *inst*) =4=> *adaptDefinition*(*p.referencedDefinition0, inst*)

c= <composite state reference>(*) =4=> *adaptDefinition*(*c.referencedDefinition0, undefined*)

s= <system type reference>(*) =4=> *adaptDefinition*(*s.referencedDefinition0, undefined*)

b= <block type reference>(*,*) =4=> *adaptDefinition*(*b.referencedDefinition0, undefined*)

p= <process type reference>(*,*) =4=> *adaptDefinition*(*p.referencedDefinition0, undefined*)

c= <composite state type reference>(*,*) =4=> *adaptDefinition*(*c.referencedDefinition0, undefined*)

Before the properties of a <system specification> are derived, each reference is replaced by the corresponding <referenced definition>. In this replacement, the <qualifier> of the <referenced definition> is removed. If the <referenced definition> is a <diagram> referenced from a <definition>, or is <definition> referenced from a <diagram>, the <referenced definition> is considered translated to the appropriate grammar during the replacement.

Auxiliary functions

The function *referencedDefinition0* finds the corresponding entity definition for a given reference.

referencedDefinition0(*ref*: *REFERENCE0*): <referenced definition> =$_{\text{def}}$
let *eKind = ref.referenceKind0* **in**
let *refName = ref.referenceName0* **in**
if ($\exists!$ *d*$\in$<referenced definition>:
isAncestorAS0(*d.parentAS0, ref*) $\wedge$*d.entityName0 = refName* $\wedge$ *d. entityKind0 =eKind*) **then**
let *d = take*({*d*$\in$<referenced definition>:
isAncestorAS0(*d.parentAS0,ref*) $\wedge$ *d.entityName0=refName* $\wedge$ *d. entityKind0 =eKind*}) **in**
if *isQualifierMatched0*(*d.qualifier0, ref.surroundingScopeUnit0*) **then** *d*
else
undefined
endif
else
undefined
endif
endlet

The function *referencedBy0* finds the corresponding reference for a <referenced definition>.

referencedBy0(*rd*: <referenced definition>): *REFERENCE0* =$_{\text{def}}$
take({ *ref*$\in$ *REFERENCE0* : *ref.referencedDefinition0 = rd*})

The function *isPathItemMatched0* is used to determine if the first path item constitutes the leftmost part of the second path item.

isPathItemMatched0(*seq1*: <path item>*, *seq2*: <path item>*): *BOOLEAN* =$_{\text{def}}$
(*seq1.length*$\leq$ *seq2.length*$\wedge$
(*i*$\in$1..*seq1.length*:

seq1[*i*].**s-**<name> = *seq2*[*i*].**s-**<name> ∧
seq1[*i*].**s-**<scope unit kind> = *seq2*[*i*].**s-**<scope unit kind>))

The function *adaptDefinition* is used to adapt an inserted referenced definition: delete the qualifiers and merge the number of instances.

adaptDefinition(*def*:<referenced definition>, *inst*:<number of instances>):<referenced definition> =$_{def}$
case *def* **of**
| <package definition>(*uses*, <package heading>(*, *name*, *intf*), *entities*)
=> <package definition>(*uses*, <package heading>(*undefined*, *name*, *intf*), *entities*)
| <internal procedure definition>(*uses*,
<procedure heading>(*h1*, *, *name*, *h2*, *h3*, *h4*, *h5*, *h6*, *h7*),
entities, *body*)
=> <internal procedure definition>(*uses*,
<procedure heading>(*h1*, *undefined*, *name*, *h2*, *h3*, *h4*, *h5*, *h6*, *h7*),
entities, *body*)
| <block definition>(*uses*, <block heading>(*, *name*, <agent instantiation>(*inst1*, *addi*)), *body*)
=> <block definition>(*uses*, <block heading>(*undefined*, *name*,
<agent instantiation>(*mergeNumbers*(*inst*, *inst1*), *addi*)), *body*)
| <process definition>(*uses*,
<process heading>(*, *name*, <agent instantiation>(*inst1*, *addi*)), *body*)
=> <process definition>(*uses*, <process heading>(*undefined*, *name*,
<agent instantiation>(*mergeNumbers*(*inst*, *inst1*), *addi*)), *body*)
| <interface definition> => *def*
| <composite state>(*uses*, <composite state heading>(*, *name*, *p*), *body*)
=> <composite state>(*uses*, <composite state heading>(*undefined*, *name*, *p*), *body*)
| <system type definition>(*uses*, <system type heading>(*, *name*, *addi*), *body*)
=> <system type definition>(*uses*, <system type heading>(*undefined*, *name*, *addi*), *body*)
| <block type definition>(*uses*, <block type heading>(*pre*, *, *name*, *addi*), *body*)
=> <block type definition>(*uses*, <block type heading>(*pre*, *undefined*, *name*, *addi*), *body*)
| <process type definition>(*uses*, <process type heading>(*pre*, *, *name*, *addi*), *body*)
=> <process type definition>(*uses*, <process type heading>(*pre*, *undefined*, *name*, *addi*), *body*)
otherwise *undefined*
endcase

The function *mergeNumbers* is used to adapt an inserted referenced definition.

mergeNumbers(*inst1*: <number of instances>, *inst2*: <number of instances>):
<number of instances> =$_{def}$
if *inst1* = *undefined* **then** *inst2*
elseif *inst2* = *undefined* **then** *inst1*
else
let *ini1* = *inst1*.s-<initial number> **in**
let *max1* = *inst1*.s-<maximum number> **in**
<number of instances>(**if** *ini1* ≠ *undefined* **then** *ini1* **else** *inst2*.s-<initial number> **endif**,
if *max1* ≠ *undefined* **then** *max1* **else** *inst2*.s-<maximum number> **endif**)
endlet
endif

The function *isFormalContextParametersMatched0* is used to determine if two <formal context parameters> match.

isFormalContextParametersMatched0(*fcpl1*: <name>*, *fcpl2*: <name>*): *BOOLEAN* =$_{def}$
if (*fcpl1* = *empty*) **then** (*fcpl2* = *empty*)
else
isFormalContextParameterMatched0(*fcpl1.head*, *fcpl1.tail*) ∧
isFormalContextParametersMatched0 (*fcpl1.tail*, *fcpl2.tail*)
endif

isFormalContextParameterMatched0(*fcp1*: <name>*, *fcp2*: <name>*): *BOOLEAN* =$_{def}$
if (*fcp1*∈<agent type context parameter>) **then** (*fcp2*∈<agent type context parameter>)
elseif (*fcp1*∈<agent context parameter>) **then** (*fcp2*∈<agent context parameter>)
elseif (*fcp1*∈<procedure context parameter>) **then** (*fcp2*∈<procedure context parameter>)

elseif (*fcp1*∈<remote procedure context parameter>) **then** (*fcp2*∈<remote procedure context parameter>)
elseif (*fcp1*∈<signal context parameter>) **then** (*fcp2*∈<signal context parameter>)
elseif (*fcp1*∈<variable context parameter>) **then** (*fcp2*∈<variable context parameter>)
elseif (*fcp1*∈<remotevariable context parameter>) **then** (*fcp2*∈<remotevariable context parameter>)
elseif (*fcp1*∈<timer context parameter list>) **then** (*fcp2*∈<timer context parameter list>)
elseif (*fcp1*∈<synonym context parameter list>) **then** (*fcp2*∈<synonym context parameter list>)
elseif (*fcp1*∈<sort context parameter>) **then** (*fcp2*∈<sort context parameter>)
elseif (*fcp1*∈<compositestate type context parameter>)
then (*fcp2*∈<compositestatetype context parameter>)
elseif (*fcp1*∈<gate context parameter>) **then** (*fcp2*∈<gate context parameter>)
elseif (*fcp1*∈<interface context parameter list>) **then** (*fcp2*∈<interface context parameter list>)
endif

F2.2.4.4 Select definition

Concrete syntax

<select definition> ::=
<Boolean><simple expression>
{<signal definition list>
| <data definition>
| <variable definition>
| <timer definition>
| <macro definition>
| <remote variable definition>
| <remote procedure definition>
| <select definition>
| <operation definition>
| <block reference>
| <process reference>
| <agent type reference>
| <procedure definition>
| <imported variable specification>
| <imported procedure specification>
| <signal list definition>
| <agent type definition>
| <agent definition>
| <channel definition>
| <channel to channel connection>
| <composite state>
| <composite state reference>
| <composite state type definition>
| <composite state type reference>
| <package definition>
| <package reference>
| <procedure reference>
| <state machine>
| <state partition> }+

Transformations

< <select definition>(*cond*, *defs*) >
=7=> **if** *value0*(*cond*) **then** *defs* **else** *empty* **endif**

The <select definition> and the <option area> are deleted at transformation and are replaced by the contained selected constructs, if any.

F2.2.4.5 Transition option

Concrete syntax

<transition option> :: <alternative question> <textual decision body>

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

Transformations

t=<transition action items>(*a*, *undefined*)
provided *a.last* ∉ <transition option> ∧ *t.parentAS0.parentAS0.parentAS0* ∈ <transition option> ∧
t.findContinueLabel ≠ *undefined*
=7=> <transition action items>(*a*,
<terminator>(*undefined*,<join>(*findContinueLabel*(*t*))))

If a <transition option> is not terminating, then it is derived syntax for a <transition option> wherein all the <textual answer part>s and the <textual else part> have inserted in their <transition>:

a) if the transition option is the last <action> in a <transition string>, a <join> to the following <terminator>; or

b) else a <join> to the first <action> following the transition option.

<transition option>(*q*, <textual decision body>(*answers*, *elsePart*))
=7=> (**let** *matching* = { *a*.**s-**<transition> | *a* ∈ *answers.toSet*: *q*. *value0* ∈ *a*.**s-**<answer> } **in**
if *matching* ≠ ∅ **then** *matching.take*
elseif *elsePart* ≠ *undefined* **then** *elsePart*.**s-**<transition>
else *undefined*
endif
endlet)

The <transition option> and <transition option area> are deleted at transformation and replaced by the contained selected constructs.

F2.2.4.6 Associations

Associations do not have semantics in SDL-2010.

F2.2.5 Structural concepts

F2.2.5.1 Structural type definitions

F2.2.5.1.1 Agent types

Abstract syntax

Agent-type-definition	::	*Agent-type-name* *Agent-kind* [*Agent-type-identifier*] *Agent-formal-parameter** *Data-type-definition*-**set** *Syntype-definition*-**set** *Signal-definition*-**set** *Timer-definition*-**set** *Variable-definition*-**set** *Agent-type-definition*-**set** *Composite-state-type-definition*-**set** *Procedure-definition*-**set** *Agent-definition*-**set** *Gate-definition*-**set** *Channel-definition*-**set** *State-machine* [*Abstract*]
Agent-kind	=	**SYSTEM** | **BLOCK** | **PROCESS**

Conditions on abstract syntax

∀*d*∈*Agent-type-definition*: *d.agentKind1* ∈ {**SYSTEM**, **PROCESS**}⇒
(*d*.**s-***Agent-definition*-**set** ≠ ∅∨ *d*.**s-***State-machine* ≠ *undefined*)

An *Agent* with the *Agent-kind* **SYSTEM** or **PROCESS** must contain either at least one *Agent-definition* or an explicit or implicit *State-machine*.

$\forall d \in$ *Agent-type-definition*: (*d.agentKind1* = **PROCESS**) $\Rightarrow$
($\forall$ *d'* $\in$ *Agent-type-definition*$\cup$*Agent-definition*: *d'.parentAS1*= *d* $\Rightarrow$ *d'*. *agentKind1*= **PROCESS**)

The contained *Agent-definition*s and *Agent-type-definition*s of a Process must all have the *Agent-kind* **PROCESS.**

Concrete syntax

<agent type definition> =
    <system type definition>
  | <block type definition>
  | <process type definition>

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

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

Transformations

a = <agent body>(*excAndStart*, *items*) =1=>
  <interaction>(*empty*,
    <composite state>(*empty*,
      <composite state heading>(*empty*, *a.parentAS0.parentAS0.name0*, *empty*),
      <composite state structure>(*undefined*, *empty*, *empty*, *empty*,
        <composite state body>(
          if *excAndStart* = *undefined* **then** *empty* **else** < *excAndStart*.**s-**<start> > **endif**,
          items
))))

An agent type with an <agent type body> or an <agent type body area> is shorthand for an agent type having only a state machine, but no contained agents. This state machine is obtained by replacing the <agent type body> or <agent type body area> by a composite state definition. This composite state definition has the same name as the agent type and its State-transition-graph is represented by the <agent type body> or the <agent type body area>.

Auxiliary functions

The function *validOutputSignalSet0* gets the complete output signal set of an <agent type definition> or an <agent definition>.

validOutputSignalSet0(*d*: <agent type definition>$\cup$<agent definition>):*SIGNAL0* =$_{\text{def}}$
  d.localOutputSignalSet0$\cup$ *d.inheritedOutputSignalSet0*

localOutputSignalSet0(*d*: <agent type definition>$\cup$<agent definition>):*SIGNAL0* =$_{\text{def}}$
  {*sid*$\in$*SIGNAL0*:
    ($\exists$*gc*$\in$<gate constraint>:
      (*gc.parentAS0*$\in$<textual gate definition>)$\wedge$*isDefinedIn0*(*gc.parentAS0*, *d*)$\wedge$
      (*gc.direction0*=**out**)$\wedge$(*sid*$\in$*signalSet0*(*gc*.**s-**<signal list item>**-seq**))) $\vee$
    ($\exists$*cp*$\in$<channel path>:
      (*cp.parentAS0*$\in$<channel definition>)$\wedge$*isDefinedIn0*(*cp.parentAS0*, *d*)$\wedge$
      (*cp.destination0*.**s-env**$\neq$*undefined*)$\wedge$(*sid*$\in$*signalSet0*(*cp*.**s-**<signal list item>**-seq**)))}

inheritedOutputSignalSet0(*d*: <agent type definition>$\cup$<agent definition>):*SIGNAL0* =$_{\text{def}}$
  if *d.specialization0 = undefined* **then** $\varnothing$
  else *d.specialization0*.**s-**<type expression>*.baseType0.validOutputSignalSet0*
  endif

F2.2.5.1.2 System type

Concrete syntax

<system type definition> ::
    <package use clause>* <system type heading> <agent structure>

<system type heading> :: <qualifier> <system<name> <agent type additional heading>

Conditions on concrete syntax

∀*fcp*∈<formal context parameter>:
  (*fcp.surroundingScopeUnit0*∈<system type definition>) ⇒
    (*fcp*∉<agent context parameter>∪<variable context parameter list>∪<timer context parameter list>)

A <formal context parameter> of <formal context parameters> must not be an <agent context parameter>, <variable context parameter list> or <timer context parameter list>.

¬(∃*fps*∈<agent formal parameters>: *fps.surroundingScopeUnit0*∈<system type definition>)

The <agent type additional heading> in a <system type definition> may not include <agent formal parameters>.

∀*s*∈ <system type definition>:∀*qual* = *s*.**s-**<system type heading>.**s-**<qualifier>:
  qual ≠ *undefined* ⇒ *isQualifierMatched0*(*qual*,*s*)

If a <system type heading> in a <system type definition> includes a <qualifier>, that <qualifier> must match the rightmost part of the full qualifier of the <system type definition>.

Mapping to abstract syntax

| <system type definition>(*, <system type heading>(*, *name*,
    <agent type additional heading>(*, *, <agent additional heading>(*spec*, *params*))),
    <agent structure>(*, *entities*, *body*))
  => **mk-***Agent-type-definition*(*Mapping*(*name*),**SYSTEM**,*Mapping*(*spec*), *Mapping*(*params*),
      { *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Data-type-definition*)},
      { *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Syntype-definition*)},
      { *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Signal-definition*)},
      { *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Timer-definition*)},
      { *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Variable-definition*)},
      { *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Agent-type-definition*)},
      { *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Composite-state-type-definition*)},
      { *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Procedure-definition*)},
      { *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Agent-definition*)},
      { *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Gate-definition*)},
      { *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Channel-definition*)},
      Mapping(*body*))

F2.2.5.1.3 Block type

Concrete syntax

<block type definition> ::
    <package use clause>* <block type heading> <agent structure>

<block type heading> ::
    <type preamble> <qualifier> <block type<name> <agent type additional heading>

Mapping to abstract syntax

| <block type definition>(*, <block type heading>(*,*, *name*,
    <agent type additional heading>(*,*,<agent additional heading>(*spec*,*params*))),
    <agent structure>(*, *entities*, *body*))
  => **mk-***Agent-type-definition*(*Mapping*(*name*),**BLOCK**,*Mapping*(*spec*), *Mapping*(*params*),
      { *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Data-type-definition*)},
      { *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Syntype-definition*)},

{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Signal-definition*)},
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Timer-definition*)},
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Variable-definition*)},
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Agent-type-definition*)},
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Composite-state-type-definition*)},
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Procedure-definition*)},
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Agent-definition*)},
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Gate-definition*)},
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Channel-definition*)},
Mapping(*body*))

F2.2.5.1.4 Process type

Concrete syntax

<process type definition> ::
<package use clause>* <process type heading> <agent structure>

<process type heading> ::
<type preamble> <qualifier> <process type<name> <agent type additional heading>

Mapping to abstract syntax

| <process type definition>(*, <process type heading>(*,*, *name*,
<agent type additional heading>(*,*,<agent additional heading>(*spec*,*params*))),
<agent structure>(*, *entities*, *body*))
=> **mk-***Agent-type-definition*(*Mapping*(*name*),**PROCESS**,*Mapping*(*spec*), *Mapping*(*params*),
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Data-type-definition*)},
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Syntype-definition*)},
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Signal-definition*)},
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Timer-definition*)},
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Variable-definition*)},
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Agent-type-definition*)},
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Composite-state-type-definition*)},
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Procedure-definition*)},
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Agent-definition*)},
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Gate-definition*)},
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Channel-definition*)},
Mapping(*body*))

F2.2.5.1.5 Composite state type

Abstract syntax

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

Conditions on abstract syntax

∀*d*∈*Composite-state-type-definition*: *d*.**s**-*Gate-definition*-**set** ≠ ∅⇒
(∃*sd*∈*State-machine*:*getEntityDefinition1*(*sd*.**s**-*Composite-state-type-identifier*, **state type**) = *d*)

The *Gate-definition-set* must be empty unless the composite state is used as a *State-machine*.

Concrete syntax

<composite state type definition> ::
    <composite state type graph> | <state aggregation type>

<composite state type graph> ::
    <package use clause>* <composite state type heading> <composite state structure> [<name>]

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

<state aggregation type> ::
    <package use clause>* <state aggregation type heading> <aggregation structure> [<name>]

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

Mapping to abstract syntax

| <composite state type definition>(*,
  <composite state type heading>(*, *, *name*, *, *, *parent*, *params*),
    <composite state structure>(*, *gates*, *conns*, *entities*, *body*= <composite state body>(*,*,*)))
=> **mk-***Composite-state-type-definition*(*Mapping*(*name*), *Mapping*(*parent*), *Mapping*(*params*),
    bigSeq(*Mapping*(< *c* **in** *conns*: (*c* ∈ <state entry points>) >)),
    bigSeq(*Mapping*(< *c* **in** *conns*: (*c* ∈ <state exit points>) >)),
    Mapping(*gates*),
    { *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Data-type-definition*)},
    { *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Syntype-definition*)},
    { *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Composite-state-type-definition*)},
    { *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Variable-definition*)},
    { *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Procedure-definition*)},
    Mapping(*body*))

| <composite state type definition>(*,
  <composite state type heading>(*, *, *name*, *, *, *parent*, *params*),
    <composite state structure>(*, *gates*, *conns*, *entities*, <state aggregation body>(*body*)))
=> **mk-***Composite-state-type-definition*(*Mapping*(*name*), *Mapping*(*parent*), *Mapping*(*params*),
    bigSeq(*Mapping*(< *c* **in** *conns*: (*c* ∈ <state entry points>)>)),
    bigSeq(*Mapping*(< *c* **in** *conns*: (*c* ∈ <state exit points>)>)),
    Mapping(*gates*),
    { *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Data-type-definition*)},
    { *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Syntype-definition*)},
    { *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Composite-state-type-definition*)},
    { *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Variable-definition*)},
    { *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Procedure-definition* ∧
                                        e.**s-***Procedure-name* ∉ {"entry","exit"})},
    mk-*State-aggregation-node*(
      < **mk-***State-partition*(*Mapping*(*b*.**s-**<typebased state partition heading>.**s-**<name>),
        Mapping(*b*.**s-**<typebased state partition heading>.**s-**<type expression>),
        < *Mapping*(*c*) | *c* **in** *body*:
          (*c* ∈ <state partition connection entry> ∧
            c.**s2-**<identifier>=*b.identifier0*)
        > ⌒
        < *Mapping*(*c*) | *c* **in** *body*:
          (*c* ∈ <state partition connection exit> ∧
            c.**s2-**<identifier> = *b.identifier0*)
        >) |
        b **in** *body*: (*b* ∈ <textual typebased state partition def>) >,
              head(< *e* **in** *Mapping*(*entities*): (*e* ∈ *Procedure-definition* ∧

e.**s**-*Procedure-name* = "entry")>),
head(< *e* **in** *Mapping*(*entities*): (*e* ∈ *Procedure-definition* ∧
e.**s**-*Procedure-name* = "exit")>)))

F2.2.5.2 Type expression

Concrete syntax

<type expression> :: <base type> [<actual context parameter>]*

<base type> =<identifier>

Conditions on concrete syntax

∀*te*∈<type expression>:
te.**s**-<actual context parameter>-**seq** ≠ *empty* ⇒ *isParameterizedType0*(*te.baseType0*)

A non-empty <actual context parameter> list is permitted if and only if <base type> denotes a parameterized type.

Transformations

let *nn*=*newName* **in**
t=<type expression>(*id*=<identifier>(*q*, *n*), *params*)
provided *params* ≠ *undefined* ∧ *params* ≠ *empty* ∧ *t.parentAS0* ∉ <specialization>
=11=> <type expression>(<identifier>(*q*, *nn*), *undefined*)
and
< *id.refersto0* >
=> < *id.refersto0*,
replaceContextParameters0(*id.refersto0.localFormalContextParameterList0, params*,
createNewType0(*nn*, *id.refersto0*) >

A <type expression> yields either the type identified by the identifier of <base type> in case there are no actual context parameters or an anonymous type defined by applying the actual context parameters to the formal context parameters of the parameterized type denoted by the identifier of <base type>.

Mapping to abstract syntax

| <type expression>(*x*, *undefined*) => *Mapping*(*x*)

Auxiliary functions

The function *isDirectSubType0* determines if a type definition is a direct subtype of an entity definition.

isDirectSubType0(*ed*: *ENTITYDEFINITION0*, *td*: *TYPEDEFINITION0*): *BOOLEAN*=$_{\text{def}}$
∃*te*∈<type expression>: *te*. *parentAS0*=*ed*. *specialization0*∧*td*=*te.baseType0*

The function *isSubtype0* determines if a type definition is a subtype of an entity definition.

isSubtype0(*sub*: *ENTITYDEFINITION0*, *sup*: *TYPEDEFINITION0*): *BOOLEAN*=$_{\text{def}}$
isDirectSubType0(*sub*, *sup*) ∨
(∃*ttd*∈ *TYPEDEFINITION0* : *isSubtype0*(*sub*, *ttd*) ∧ *isSubtype0*(*ttd*, *sup*))

The function *baseType0* is used to get the base type definition for a type expression.

baseType0(*te*: <type expression>): *TYPEDEFINITION0* =$_{\text{def}}$
getEntityDefinition0(*te*.**s**-<base type>, *te.baseTypeKind0*)

The function *baseTypeKind0* is used to get the base type kind for a type expression.

baseTypeKind0(*te*: <type expression>): *ENTITYKIND0*=$_{\text{def}}$
case *te.parentAS0* **of**
| <specialization>∪<data type specialization>∪<interface specialization> =>
if (*te.surroundingScopeUnit0*∈<agent definition>) **then agent type**

```
            else te.surroundingScopeUnit0.entityKind0
        | <procedure call body>=> procedure
        | <typebased system heading>=> system type
        | <typebased block heading>=> block type
        | <typebased process heading>=> process type
        | <typebased composite state>∪<typebased state partition heading> => state type
        otherwise undefined
        endcase
```

The function *isParameterizedType0* is used to determine if a type definition is a parameterized type.

```
isParameterizedType0(td: TYPEDEFINITION0 ): BOOLEAN=def
    (td.formalContextParameterList0 ≠ empty)
```

Get the formal context parameter list of a type definition.

```
formalContextParameterList0(td: TYPEDEFINITION0 ): <name>* =def
    td.inheritedFormalContextParameterList0⌒ td.localFormalContextParameterList0
```

Get the formal context parameter list of the super type of a type definition.

```
inheritedFormalContextParameterList0(td: TYPEDEFINITION0 ): <name>* =def
    let sp=td.specialization0 in
        if sp = undefined then empty else
        case sp of
        | <interface specialization> =>
            < getUnboundFormalContextParameterList0(tel.baseType0.formalContextParameterList0,
                tel.actualContextParameterList0)
            | tel in sp.s-<type expression>-seq >
        otherwise
            let fcpl=sp.s-<type expression>.baseType0.formalContextParameterList0in
                let acpl=sp.s-<type expression>.actualContextParameterList0 in
                getUnboundFormalContextParameterList0(fcpl, acpl)
            endlet
        endcase
        endif
    endlet
```

Get the unbound formal context parameter list of a formal context parameter list according to an actual context parameter list.

```
getUnboundFormalContextParameterList0(fcpl: <name>*, acpl: <identifier>*):<name>* =def
    if (fcpl = empty) then empty
    elseif (acpl.head= undefined) then
        <fcpl.head>⌒getUnboundFormalContextParameterList0(fcpl.tail, acpl.tail)
    else
        getUnboundFormalContextParameterList0(fcpl.tail, acpl.tail)
    endif
```

Insert the original context parameter for the unbound ones.

```
completeFormalContextParameter0(fcpl: <name>*, acpl: <identifier>*):<name>* =def
    if (fcpl = empty) then empty
    elseif (acpl.head= undefined) then
        <fcpl.head>⌒completeFormalContextParameter0(fcpl.tail, acpl.tail)
    else
        <acpl.head>⌒completeFormalContextParameter0(fcpl.tail, acpl.tail)
    endif
```

Get the actual context parameter list of a type expression.

```
actualContextParameterList0(te: <type expression>):<identifier>* =def
```

te.**s**-<actual context parameter>**-seq**

Get the formal context parameter list local to a type definition.

localFormalContextParameterList0(*td*: *TYPEDEFINITION0*): <name>* =$_{def}$
let *fcps* =*take*({*fcps*∈<agent type additional heading> ∪ <composite state type heading> ∪
<procedure heading> ∪ <signal definition> ∪ <data type heading> ∪
<interface heading>: *fcps*.*surroundingScopeUnit0* = *td*}) **in**
< *fcpl*.*formalContextParameterSublist0* | *fcpl* **in** *fcps*.**s**-<formal context parameter>**-seq** >
endlet

Get the list of names made up of a formal context parameter.

formalContextParameterSublist0(*fcp*: <formal context parameter>):<name>* =$_{def}$
case *fcp* **of**
| <agent type context parameter> ∪ <agent context parameter>=> <*fcp*.**s**-<name>>
| <procedure context parameter>∪<remote procedure context parameter> => <*fcp*.**s**-<name>>
| <signal context parameter list> =>
< *scpl*.**s**-<name> | *scpl* **in** *fcp*.**s**-<signal context parameter name>**-seq** >
| <variable context parameter list> =>
< *vcpl*.**s**-<name> | *vcpl* **in** *fcp*.**s**-<variable context parameter names>**-seq** >
| <remotevariable context parameter list> =>
< *rvcpl*.**s**-<name> | *rvcpl* **in** *fcp*.**s**-<remotevariable contextparameter names>**-seq**>
| <timer context parameter list> =>
< *tcpl*.**s**-<name> | *tcpl* **in** *fcp*.**s**-<timer context parameter name>**-seq**>
| <synonym context parameter list> =>
< *scpl*.**s**-<name> | *scpl* **in** *fcp*.**s**-<synonym context parameter name>**-seq**>
| <sort context parameter> => <*fcp*.**s**-<name>>
| <compositestate type context parameter> => <*fcp*.**s**-<name>>
| <gate context parameter> => <*fcp*.**s**-<gate>.**s**-<name>>
| <interface context parameter list> =>
< *icpl*.**s**-<name> | *icpl* **in** *fcp*.**s**-<interface context parameter name>**-seq**>
otherwise *empty*
endcase

Replace the context parameters by their values.

replaceContextParameters0(*p*: *DefinitionAS0**, *v*: *DefinitionAS0**, *orig*: *DefinitionAS0*):
DefinitionAS0 =$_{def}$
if *p* = *empty* **then** *orig*
else *replaceContextParameters0*(*p.tail*, *v.tail*, *replaceInSyntaxTree*(*p.head*, *v.head*, *orig*))
endif

Create a new type without any formal context parameters.

createNewType0(*n*: <name>, *orig*: *DefinitionAS0*): *DefinitionAS0* =$_{def}$
case *orig* **of**
| <system type definition>(*use*,
<system type heading>(*q*, *, <agent type additional heading>(*, *virt*, *add*)), *body*)
=> <system type definition>(*use*,
<system type heading>(*q*, *n*, <agent type additional heading>(*empty*, *virt*, *add*)), *body*)
| <block type definition>(*use*,
<block type heading>(*pre*, *q*, *, <agent type additional heading>(*, *virt*, *add*)), *body*)
=> <block type definition>(*use*,
<block type heading>(*pre*, *q*, *n*, <agent type additional heading>(*empty*, *virt*, *add*)), *body*)
| <process type definition>(*use*,
<process type heading>(*pre*, *q*, *, <agent type additional heading>(*, *virt*, *add*)), *body*)
=> <process type definition>(*use*,
<process type heading>(*pre*, *q*, *n*, <agent type additional heading>(*empty*, *virt*, *add*)), *body*)
| <composite state type definition>(*use*,
<composite state type heading>(*v*, *q*, *, *, *c*, *spec*, *par*), *body*)
=> <composite state type definition>(*use*,
<composite state type heading>(*v*, *q*, *n*, *empty*, *c*, *spec*, *par*), *body*)
| <data type definition>(*use*, *pre*, <data type heading>(*k*, *, *, *v*), *spec*, *body*)

=> <data type definition>(*use*, *pre*, <data type heading>(*k*, *n*, *empty*, *v*), *spec*, *body*)
| <internal procedure definition>(*use*, <procedure heading>(*pre*, *q*, *, *, *c*, *spec*, *par*, *res*), *ent*, *body*)
=> <internal procedure definition>(*use*,
<procedure heading>(*pre*, *q*, *n*, *empty*, *c*, *spec*, *par*, *res*), *ent*, *body*)
| <interface definition>(*use*, *virt*, <interface heading>(*, *, *v*), *spec*, *ent*, *l*)
=> <interface definition>(*use*, *virt*, <interface heading>(*n*, *empty*, *v*)), *spec*, *ent*, *l*)
| <signal definition>(*, *, *v*, *spec*, *l*)
=> <signal definition>(*n*, *empty*, *v*, *spec*, *l*)
otherwise *undefined*
endcase

F2.2.5.3 Definitions based on types

Concrete syntax

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

Conditions on concrete syntax

∀*ad*∈<textual typebased agent definition>: ∀*te*∈<type expression>:
(*te.parentAS0.parentAS0= ad*) ⇒
(∃*s*∈<start>: (*s*∈*te.baseType0.startSet0*)∧(*s*.**s-**<name> = *undefined*))

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

F2.2.5.3.1 System definition based on system type

Concrete syntax

<textual typebased system definition>::
<typebased system heading>

<typebased system heading> ::
<system<name> <system<type expression>

Mapping to abstract syntax

| <textual typebased system definition>(<typebased system heading>(*name*,<type expression>(*b*,*)))
=> **mk-***Agent-definition*(*Mapping*(*name*), **mk-***Number-of-instances*(1,1), *Mapping*(*b*))

F2.2.5.3.2 Block definition based on block type

Concrete syntax

<textual typebased block definition> ::
<typebased block heading>

<typebased block heading> ::
<block<name> <number of instances> <block<type expression>

Mapping to abstract syntax

| <textual typebased block definition>
(<typebased block heading>(*name*, *inst*, <type expression>(*b*,*)))
=> **mk-***Agent-definition*(*Mapping*(*name*), *Mapping*(*inst*), *Mapping*(*b*))

F2.2.5.3.3 Process definition based on process type

Concrete syntax

<textual typebased process definition> = <typebased process heading>

<typebased process heading> ::
<process<name> <number of instances> <process<type expression>

Mapping to abstract syntax

| <textual typebased process definition> (*name*, *inst*, <type expression>(*b*,*))
=> **mk-***Agent-definition*(*Mapping*(*name*), *Mapping*(*inst*), *Mapping*(*b*))

F2.2.5.3.4 Composite state definition based on composite state type

Concrete syntax

<typebased composite state> :: <composite state name> <composite state<type expression>

<textual typebased state partition def> =
<typebased state partition heading>

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

Mapping to abstract syntax

A composite state based on a type is mapped within the production for states.

F2.2.5.4 Abstract type

Further study is needed. For example, mapping to the abstract syntax is not defined and a data type cannot be instantiated. The abstract property of a type is not inherited; therefore instantiation of a subtype of an abstract data type is permitted, if the subtype is not itself abstract (see clause 8.1.3 of [ITU-T Z.102]).

Abstract syntax

Abstract :: {}

Concrete syntax

<abstract> :: ()

Conditions on concrete syntax

$\forall pd \in$ <procedure definition>: *isAbstractType0*(*pd*) $\Rightarrow$
$\neg(\exists pc \in$ <procedure call>: *pd* = *pc.calledProcedure0*)

An abstract procedure cannot be called.

$\forall ad \in$ <textual typebased agent definition>: $\forall te \in$ <type expression>:
te.parentAS0.parentAS0= ad $\Rightarrow \neg$*isAbstractType0*(*te.baseType0*)

A typebased agent shall not be specified with an abstract agent type as the type.

$\forall td \in$ *TYPEDEFINITION0* : *isAbstractType0*(*td*) $\Rightarrow$
$\neg(\exists d \in$ <textual typebased agent definition> $\cup$ <textual typebased state partition def> $\cup$
<typebased composite state>:$\exists te \in$ <type expression>:
((*te.parentAS0.parentAS0= d*)$\vee$(*te.parentAS0= d*))$\wedge$ (*te.baseType0=td*))

An abstract type cannot be instantiated.

Auxiliary functions

Determine if a type definition is abstract, either because it is defined as abstract by the keyword **abstract** or because it has unbound context parameters.

isAbstractType0(*td*: *TYPEDEFINITION0*): *BOOLEAN*$=_{\text{def}}$
($\exists ab \in$ <abstract>: *ab.surroundingScopeUnit0 = td*)
$\vee$ ($\exists te \in$ <abstract>: *te.baseType0= td* $\wedge$
getUnboundFormalContextParameterList0(*te.baseType0.formalContextParameterList0,*
te.actualContextParameterList0) $\neq$ *empty*)

Get the <internal procedure definition> denoted by a <procedure call>.

calledProcedure0(*pc*: <procedure call>$\cup$<value returning procedure call>

∪<procedure call body>∪<remote procedure call body>):
<procedure definition>=$_{def}$
case *pc* **of**
| <procedure call>∪<value returning procedure call>=>
let *t* = *pc*.**s-**<procedure call body>.**s-implicit in**
if *t*∈<identifier> **then** *getEntityDefinition0*(*t*, **procedure**)
else *t.baseType0* // *t*∈<type expression>
| <procedure call body>=>
let *t* = *pc*.**s-implicit in**
if *t*∈<identifier> **then** *getEntityDefinition0*(*t*, **procedure**)
else *t.baseType0* // *t*∈<type expression>
| <remote procedure call body>=> *getEntityDefinition0*(*pc*, **remote procedure**)
| **otherwise** => *undefined*
endcase

F2.2.5.5 Type reference

Type references do not have semantics in SDL-2010.

F2.2.5.6 Gate

Abstract syntax

Gate-definition	::	*Gate-name* [*Encoding-rules*] *In-signal-identifier*-**set** *Out-signal-identifier*-**set**
In-signal-identifier	=	*Signal-identifier*
Out-signal-identifier	=	*Signal-identifier*

Concrete syntax

<gate in definition> = <textual gate definition> | <textual interface gate definition>

<textual gate definition> ::
<gate> [<encoding rules>] <gate constraint>

<textual interface gate definition> ::
{ **out** | **in** } <interface<identifier> [<encoding rules>] [<textual endpoint constraint>]

<gate> = <gate<name>

<gate constraint> ::
in [<textual endpoint constraint>] [<signal list>]
[**out** [<textual endpoint constraint>] [<signal list>]]
| **out** [<textual endpoint constraint>] [<signal list>]

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

<as gate> :: <gate<identifier>

Further study is needed for the handling of <as gate>.

Conditions on concrete syntax

∀*te*∈<type expression>:
(*te.parentAS0*∈<typebased composite state>) ∨
(*te.parentAS0.parentAS0*∈<textual typebased agent definition>∪
<textual typebased state partition def>) ⇒
(∀*gc*∈<gate constraint>:
(*gc.parentAS0*∈<textual gate definition>)∧*isDefinedIn0*(*gc.parentAS0*, *te.baseType0*) ⇒
gc.**s-**<signal list item>**-seq** ≠ *empty*)

Types from which instances are defined must have a signal list in the <gate constraint>s.

∀*gd*∈<textual gate definition>: ∀*gc*∈<gate constraint>:
(*gc.parentAS0*= *gd*) ⇒

(**let** *td = gd.surroundingScopeUnit0* **in**
(∃*td'*∈*TYPEDEFINITION0* :
td' = getEntityDefinition0(*gc*.**s**-<textual endpoint constraint>, *td.entityKind0*)) **endlet**)

The <identifier> of <textual endpoint constraint> must denote a type definition of the same entity kind as the type definition in which the gate is defined.

∀*gc* ∈ <gate constraint>:(∀*ce* ∈ <channel endpoint>:(∀*ce'* ∈ <channel endpoint>:(
(*gc.parentAS0*.**s**-<gate> = *ce*.**s**-<gate>) ∧
(*ce* ≠ *ce'*) ∧ (*ce.parentAS0= ce'.parentAS0*) ∧ (*ce.parentAS0*∈<channel path>) ∧
(*gc.parentAS0*∈<textual gate definition>))))⇒
(**let** *td = getEntityDefinition0*(*gc*.**s**-<textual endpoint constraint>,
gc.surroundingScopeUnit0.entityKind0) **in**
∃*td'* ∈ *ENTITYDEFINITION0*:
((*td' = ce'.channelEndpointReferTo0*) ∧ ((*td = td'*) ∨ *isSubtype0*(*td', td*)))
endlet) ∧
(*ce.parentAS0*.**s**-<signal list item>-**seq**.*signalSet0* ⊆ *gc*.**s**-<signal list item>-**seq**.*signalSet0*)

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

∀*tbd*∈<textual typebased block definition> ∪<textual typebased process definition>:
∀*te*∈<type expression>: (*te.parentAS0.parentAS0= tbd*) ⇒
(**let** *td = te.baseType0* **in**
(*td.channelDefinitionSet0* ≠ ∅) ⇒
(∀*gc*∈<gate constraint>:∀*sig* ∈ *SIGNAL0*:
(*gc.parentAS0*∈*td.gateDefinitionSet0*) ∧
(*sig*∈*gc*.**s**-<signal list item>-**seq**.*signalSet0*) ⇒
(∃*cp*∈<channel path>:
(*cp.parentAS0*∈*td.channelDefinitionSet0*) ∧
((*gc.direction0*=**in**)⇒
(*cp.origination0*.**s**-<gate> = *gc.parentAS0*.**s**-<gate>) ∧
(*cp.origination0*.**s-implicit**=**env**)) ∧
((*gc.direction0*=**out**)⇒
(*cp.destination0*.**s**-<gate> = *gc.parentAS0*.**s**-<gate>) ∧
(*cp.destination0*.**s-implicit**=**env**)) ∧
(*sig*∈*cp*.**s**-<signal list item>-**seq**.*signalSet0*))) **endlet**)

If the type denoted by <base type> in a <textual typebased block definition> or <textual typebased process definition> contains channels, the following rule applies: For each combination of (gate, signal, direction) defined by the type, the type must contain at least one channel that - for the given direction - mentions **env** and the gate and either mentions the signal or has no explicit <signal list> associated. In the latter case, it must be possible to derive that the channel is able to carry the signal in the given direction. If the type contains channels mentioning remote procedures or remote variables, a similar rule applies.

∀*gc, gc'*∈<gate constraint>:
(*gc* ≠ *gc'*) ∧ (*gc.parentAS0= gc'.parentAS0*) ⇒
(*gc*.**s**-<textual endpoint constraint> = *gc'*.**s**-<textual endpoint constraint>) ∧
((*gc.direction0*=**out**) ∧ (*gc'.direction0*=**in**)∨(*gc.direction0*=**in**)∧(*gc'.direction0*=**out**))

Where two <gate constraint>s are specified one must be in the reverse direction to the other, and the <textual endpoint constraint>s of the two <gate constraint>s must be the same.

∀*gd*∈<textual gate definition>: (*gd*.**s-adding** ≠ *undefined*) ⇒
(**let** *td = gd.surroundingScopeUnit0* **in**
∃*td'*∈*TYPEDEFINITION0* : ∃*gd'*∈<textual gate definition>:
isSubtype0(*td, td'*) ∧ (*gd'*∈ *td'.gateDefinitionSet0*) ∧

(*gd'*.**s**-<gate> = *gd*.**s**-<gate>.) **endlet**)

adding may only be specified in a subtype definition and only for a gate defined in the supertype.

∀*ec*,*ec'*∈<textual endpoint constraint>:
isSubtype0(*ec.surroundingScopeUnit0*, *ec'.surroundingScopeUnit0*) ∧
(*ec.parentAS0*∈<gate constraint>) ∧ (*ec'.parentAS0*∈<gate constraint>) ∧
(*ec.parentAS0.parentAS0*∈<textual gate definition>) ∧
(*ec'.parentAS0.parentAS0*∈<textual gate definition>) ∧
(*ec.parentAS0.parentAS0*.**s-adding** ≠ *undefined*) ∧
(((*ec.direction0*=**out**) ∧ (*ec'.direction0*=**in**))∨((*ec.direction0*=**in**) ∧ (*ec'.direction0*=**out**))) ∧
(*ec.parentAS0.parentAS0*.**s**-<gate>= *ec'.parentAS0.parentAS0*.**s**-<gate>) ⇒
(**let** *td* = *getEntityDefinition0*(*ec*.**s**-<identifier>, *ec.surroundingScopeUnit0.entityKind0*) **in**
let *td'* = *getEntityDefinition0*(*ec'*.**s**-<identifier>, *ec'.surroundingScopeUnit0.entityKind0*) **in**
(*td* = *td'*) ∨ *isSubtype0*(*td*, *td'*) **endlet**)

If <textual endpoint constraint> is specified for the gate in the supertype, the <identifier> of an (added) <textual endpoint constraint> must denote the same type or a subtype of the type denoted in the <textual endpoint constraint> of the supertype.

∀*gd*∈<textual interface gate definition>:∃*td*∈<interface definition>:
td = *getEntityDefinition0*(*gd*.**s**-<identifier>, **interface**)

The <interface identifier> of a <textual interface gate definition> must not identify the interface implicitly defined by the entity to which the gate is connected.

Transformations

t= <textual interface gate definition>(**out**, *id*=<identifier>(*q*,*n*)) =1=>
<textual gate definition>(*n*, *undefined*,
<gate constraint>(**out**, <textual endpoint constraint>(*undefined*, *id*), *empty*))

t= <textual interface gate definition>(**in**, *id*=<identifier>(*q*,*n*)) =1=>
<textual gate definition>(*n*, *undefined*,
<gate constraint>(**in**, <textual endpoint constraint>(*undefined*, *id*), *empty*))

<textual interface gate definition> and <graphical interface gate definition> are shorthand for a <textual gate definition> or a <graphical gate definition>, respectively, having the name of the interface as <gate name> and the <interface identifier> as the <gate constraint> or <signal list area>.

Mapping to abstract syntax

| <textual gate definition>(*name*,
<gate constraint>(**in**, *, *inlist*),
<gate constraint>(**out**, *, *outlist*))
=> **mk**-*Gate-definition*(*Mapping*(*name*), *Mapping*(*inlist*).*toSet*, *Mapping*(*outlist*).*toSet*)

| <textual gate definition>(*name*,
<gate constraint>(**out**, *, *outlist*),
<gate constraint>(**in**, *, *inlist*))
=> **mk**-*Gate-definition*(*Mapping*(*name*), *Mapping*(*inlist*).*toSet*, *Mapping*(*outlist*).*toSet*)

| <textual gate definition>(*name*, <gate constraint>(**in**, *, *inlist*), *undefined*)
=> **mk**-*Gate-definition*(*Mapping*(*name*), *Mapping*(*inlist*).*toSet*, ∅)

| <textual gate definition>(*name*, <gate constraint>(**out**, *, *outlist*), *undefined*)
=> **mk**-*Gate-definition*(*Mapping*(*name*), ∅, *Mapping*(*outlist*).*toSet*)

Auxiliary functions

Get the <gate in definition> defined in an <agent type definition>, a <composite state type definition>, a <agent definition> or a <composite state>.

gateDefinitionSet0(*td*: <agent type definition> ∪<composite state type definition>∪
<agent definition> ∪<composite state>): <gate in definition>-**set** =$_{def}$
td.localGateDefinitionSet0∪*td.inheritedGateDefinitonSet0*

localGateDefinitionSet0(*td*: <agent type definition> ∪ <composite state type definition> ∪
<agent definition> ∪<composite state>): <gate in definition>-**set** =$_{def}$
{*gd*∈<gate in definition>: *gd.surroundingScopeUnit0 = td*}

inheritedGateDefinitonSet0(*td*: <agent type definition> ∪ <composite state type definition>∪
<agent definition> ∪<composite state>): <gate in definition>-**set** =$_{def}$
let *sp=td.specialization0* **in**
if *sp=undefined* **then** ∅
else *sp*.**s**-<type expression>.*baseType0.gateDefinitionSet0*
endif
endlet

Get the <channel definition> defined in an <agent type definition> or a <agent definition>.

channelDefinitionSet0(*td*: <agent type definition> ∪<agent definition>): <channel definition>-**set** =$_{def}$
td.localChannelDefinitionSet0∪ *td.inheritedChannelDefinitionSet0*

localChannelDefinitionSet0(*td*: <agent type definition> ∪<agent definition>):
<channel definition>-**set** =$_{def}$
{*cd*∈<channel definition>: *cd.surroundingScopeUnit0 = td*}

inheritedChannelDefinitionSet0(*td*:<agent type definition>∪<agent definition>):
<channel definition>-**set** =$_{def}$
let *sp=td.specialization0* **in**
if *sp=undefined* **then** ∅
else *sp*.**s**-<type expression>.*baseType0.channelDefinitionSet0*
endif
endlet

Get the identifiers of the kind *SIGNAL0*.

signalSet0(*sl*:<signal list item>*): *SIGNAL0*=$_{def}$
case *sl.head.idKind0* **of**
| {**signal**, **timer**, **remote procedure**, **remote variable**}=>{*sl.head*}∪ *sl.tail.signalSet0*
| {**interface**} =>
let *fd = getEntityDefinition0*(*sl.head*, **interface**) **in**
fd.usedSignalSet0∪{*sd.identifier0*: *sd*∈ *fd.definedSignalSet0*}∪ *sl.tail.signalSet0*
| {**signallist**} =>.**s**-<signal list item>-**seq**.*signalSet0*∪
(**let** *sld = getEntityDefinition0*(*sl.head*, **signallist**) **in**
signalSet0(*sld*.**s**-<signal list item>-**seq**)∪*sl.tail.signalSet0* **endlet**)
| **otherwise** =>∅
endcase

F2.2.5.7 Context parameters

Concrete syntax

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

<formal context parameters> = <formal context parameter list>

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

<formal context parameter> =
<agent type context parameter>
| <agent context parameter>
| <procedure context parameter>
| <remote procedure context parameter>
| <signal context parameter list>
| <variable context parameter list>

| <remotevariable context parameter list>
| <timer context parameter list>
| <synonym context parameter list>
| <sort context parameter>
| <compositestate type context parameter>
| <gate context parameter>
| <interface context parameter list>

Conditions on concrete syntax

∀*fcp*∈ *FORMALCONTEXTPARAMETER0*:∀*acp*∈<actual context parameter>:
isContextParameterCorresponded0(*fcp*, *acp*) ∧ (*acp*∈<primary>) ⇒
(*fcp*∈<synonym context parameter name>)

An <actual context parameter> shall not be a <constant primary> unless it is for a synonym context parameter.

(∀*te*∈<type expression>: *te.baseType0*∉*FORMALCONTEXTPARAMETER0*) ∧
(∀*fcp*∈ *FORMALCONTEXTPARAMETER0*:
fcp.contextParameterAtleastDefinition0∉ *FORMALCONTEXTPARAMETER0*)

Formal context parameters can neither be used as <base type> in <type expression> nor in **atleast** constraints of <formal context parameters>.

∀*fcp*∈<agent type context parameter>∪<agent context parameter>∪<procedure context parameter>∪
<signal context parameter name> ∪<sort context parameter>∪
<compositestate type context parameter> ∪<interface context parameter name>:
∀*acp*∈<actual context parameter>: *isContextParameterCorresponded0*(*fcp*, *acp*) ⇒
(∀*td'*∈*TYPEDEFINITION0* : (*td'* = *fcp.contextParameterAtleastDefinition0*) ⇒
(*td'*∉*FORMALCONTEXTPARAMETER0*) ∧ ¬(*isParameterizedType0*(*td'*)) ∧
(∃*td*∈ *TYPEDEFINITION0* :
(*td* = *getEntityDefinition0*(*acp*, *td'.entityKind0*) ∧
((*td* = *td'*) ∨ *isSubtype0*(*td*, *td'*))))

An **atleast** clause denotes that the formal context parameter must be replaced by an actual context parameter, which is the same type or a subtype of the type identified in the **atleast** clause. Identifiers following the keyword **atleast** in this clause must identify type definitions of the entity kind of the context parameter and must be neither formal context parameters nor parameterized types.

Transformations

<composite state type heading>(*v*, *q*, *n*, *cPar*, *vc*,
<specialization>(<type expression>(*base*, *actPar*), *), *p*)
provided
getUnboundFormalContextParameterList0
(*actParams*, *base.refersto0*.**s-**<formal context parameter>) ≠ *empty*
=11=>
(**let** *nCPar* = *cPar* ⁀
getUnboundFormalContextParameterList0
(*actParams*, *base.refersto0*.**s-**<formal context parameter>)
in
let *nActPar* =
completeFormalContextParameter0(*actParams*, *base.refersto0*.**s-**<formal context parameter>)
in
<composite state type heading>
(*v*, *q*, *n*, *nCPar*, *vc*, <specialization>(<type expression>(*b*, *nActPar*), *undefined*), *p*)
endlet)

<agent type additional heading>(*cPar*, *vc*,
<agent additional heading>(<specialization>(<type expression>(*base*, *actPar*), *), *p*))
provided
getUnboundFormalContextParameterList0
(*actParams*, *base.refersto0*.**s-**<formal context parameter>) ≠ *empty*

```
=11=>
    (let nCPar = cPar⌒
        getUnboundFormalContextParameterList0
            (actParams, base.refersto0.s-<formal context parameter>)
    in
    let nActPar =
        completeFormalContextParameter0(actParams, base.refersto0.s-<formal context parameter>)
    in
        <agent type additional heading>(nCPar, vc,
            <agent additional heading>
                (<specialization>(<type expression>(base, nActPar), undefined), p))
    endlet)

<procedure heading>(v, q, n, cPar, vc, <specialization>(<type expression>(base, actPar), *), p, r, x)
provided
    getUnboundFormalContextParameterList0
        (actParams, base.refersto0.s-<formal context parameter>)
        ≠ empty
=11=>
    (let nCPar = cPar⌒
        getUnboundFormalContextParameterList0
            (actParams, base.refersto0.s-<formal context parameter>)
    in
    let nActPar =
        completeFormalContextParameter0(actParams, base.refersto0.s-<formal context parameter>)
    in
        <procedure heading>(v, q, n, nCPar, vc,
            <specialization>(<type expression>(b, nActPar), undefined), p, r, x)
    endlet)

<signal definition>(n, cPar, vc, <specialization>(<type expression>(base, actPar), *), p)
provided
    getUnboundFormalContextParameterList0
        (actParams, base.refersto0.s-<formal context parameter>) ≠ empty
=11=>
    (let nCPar = cPar⌒
        getUnboundFormalContextParameterList0
            (actParams, base.refersto0.s-<formal context parameter>)
    in
    let nActPar =
        completeFormalContextParameter0(actParams, base.refersto0.s-<formal context parameter>)
    in
    <signal definition>
        (n,nCPar, vc, <specialization>(<type expression>(b, nActPar), undefined*), p)
    endlet)
```

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

Auxiliary functions

Get the entity definition referred by the formal context parameter constraint.

```
contextParameterAtleastDefinition0(fcp: FORMALCONTEXTPARAMETER0): ENTITYDEFINITION0 =def
    case fcp of
    | <agent type context parameter> =>
        if (fcp.s-<agent type constraint>∈ <identifier>) then
            getEntityDefinition0(fcp.s-<agent type constraint>.<identifier>, agent type)
```

else *undefined*
endif
| <agent context parameter> =>
if (*fcp*.**s**-<agent constraint>∈<agent constraint atleast>) **then**
getEntityDefinition0(*fcp*.**s**-<agent constraint>.**s**-<identifier>, **agent type**)
else if (*fcp*.**s**-<agent constraint>∈<agent constraint exactly>) **then**
getEntityDefinition0(*fcp*.**s**-<agent constraint>.**s**-<identifier>, **agent type**)
else *undefined*
endif
endif
| <procedure context parameter> =>
if (*fcp*.**s**-<procedure constraint>∈<identifier>) **then**
getEntityDefinition0(*fcp*.**s**-<procedure constraint>, **procedure**)
else *undefined*
endif
| <compositestate type context parameter> =>
if (*fcp*.**s**-<composite state type constraint>∈<identifier>) **then**
getEntityDefinition0(*fcp*.**s**-<composite state type constraint>,
state type)
else *undefined*
endif
| <signal context parameter name> =>
if (*fcp*.**s**-<signal constraint>∈<identifier>) **then**
getEntityDefinition0(*fcp*.**s**-<signal constraint>, **signal**)
else *undefined*
endif
| <sort context parameter> =>
if (*fcp*.**s**-<sort constraint>∈<sort>) **then**
getEntityDefinition0(*fcp*.**s**-<sort constraint>, **type**)
else *undefined*
endif
| <interface context parameter name> =>
if (*fcp*.**s**-<interface constraint> ≠ *undefined*) **then**
getEntityDefinition0(*fcp*.**s**-<interface constraint>.**s**-<identifier>, **interface**)
else *undefined*
endif
otherwise *undefined*
endcase

F2.2.5.7.1 Agent type context parameter

Concrete syntax

<agent type context parameter> ::
<agent kind> <agent type<name> [<agent type constraint>]

<agent kind> :: **process** | **block**

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

Conditions on concrete syntax

∀*fcp*∈<agent type context parameter>:∀*acp*∈<actual context parameter>:
isContextParameterCorresponded0(*fcp*, *acp*) ∧ (*fcp*.**s**-<agent type constraint> ≠ *undefined*) ⇒
(**let** *td=getEntityDefinition0*(*acp*, **agent type**) **in**
(∃*td'*∈<agent type definition>:
(*td'=fcp.contextParameterAtleastDefinition0*) ∧
(*td.agentLocalFormalParameterList0=empty*) ∧ *isSubtype0*(*td*,*td'*)) ∨
((*td.entityKind0* = *fcp.entityKind0*) ∧
(**let** *pl* = *td.agentFormalParameterList0* **in**
let *sl* = *fcp*.**s**-<agent type constraint>.*agentSignatureSortList0* **in**
(*pl.length* = *sl.length*) ∧
(∀*i*∈1..*pl.length*: *isSameSort0*(*pl*[*i*].*parentAS0*.**s**-<sort>, *sl*[*i*])) **endlet**)) **endlet**)

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

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

Auxiliary functions

Get the sort list defined in an <agent signature>.

agentSignatureSortList0(*as*: <agent signature>):<sort>* =$_{def}$ (*as*.**s-**<sort>**-seq**)

Get the formal parameter list of an <agent type definition>, an <agent definition>, a <composite state type definition> or a <composite state>.

agentFormalParameterList0(*td*: <agent type definition> ∪<agent definition> ∪
<composite state type definition>∪<composite state>): <name>* =$_{def}$
td.agentLocalFormalParameterList0⌒ *td.agentInheritedFormalParameterList0*

agentLocalFormalParameterList0(*td*: <agent type definition> ∪<agent definition> ∪
<composite state type definition>∪<composite state>): <name>* =$_{def}$
let *fp=take*({*fp*∈ <agent additional heading>∪
<composite state type heading>∪
<composite state heading>:
fp.surroundingScopeUnit0 = td}) **in**
if (*fp*.**s-**<agent formal parameters>= *undefined*)
then *empty*
else < *psl*.**s-**<name> | *psl* **in** *fp*.**s-**<agent formal parameters>.<parameters of sort>**-seq** >
endif
endlet

agentInheritedFormalParameterList0(*td*: <agent type definition> ∪<agent definition> ∪
<composite state type definition>∪<composite state>): <name>* =$_{def}$
let *sp=td.specialization0* **in**
if (*sp = undefined*) **then** *empty*
else *sp*.**s-**<type expression>.*baseType0.agentFormalParameterList0*
endif
endlet

Determine if a formal context parameter corresponds to an actual context parameter.

isContextParameterCorresponded0(*fcp*: *FORMALCONTEXTPARAMETER0*,
acp: <actual context parameter>): *BOOLEAN*=$_{def}$
let *fcpl = fcp.surroundingScopeUnit0.formalContextParameterList0***in**
let *acpl = parentAS0ofKind*(*acp*, <type expression>).*actualContextParameterList0* **in**
(*fcpl.length= acpl.length*)∧
(∃!*i*∈1..*fcpl.length*: (*fcpl*[*i*]=*fcp*)∧(*acpl*[*i*]=*acp*)
endlet

F2.2.5.7.2 Agent context parameter

Concrete syntax

<agent context parameter> ::
<agent kind> <agent<name> [<agent constraint>]

<agent constraint> = <agent constraint atleast> | <agent constraint exactly> | <agent signature>

<agent constraint atleast> :: <agent type<identifier>

<agent constraint exactly> :: <agent type<identifier>

<agent signature> :: <sort list>

Conditions on concrete syntax

∀*fcp*∈<agent context parameter>: ∀*acp*∈<actual context parameter>:
isContextParameterCorresponded0(*fcp*, *acp*) ⇒
(**let** *td* = *getEntityDefinition0*(*acp*, **agent**) **in**
let *td'* = *fcp.contextParameterAtleastDefinition0* **in**
((*fcp*.**s**-<agent constraint>∈ <agent constraint atleast>)) ⇒
isSubtype0(*td*, *td'*) ∧ (*td.agentLocalFormalParameterList0* = *empty*)) ∧
((*fcp*.**s**-<agent constraint> ∈ <agent constraint exactly>)) ⇒ (*td* = *td'*)) ∧
((*fcp*.**s**-<agent constraint> ∈ <agent signature>) ⇒
(*getEntityDefinition0*(*acp*, **agent**).*entityKind0* = *fcp*.*entityKind0*) ∧
(**let** *pl* = *td.agentFormalParameterList0* **in**
let *sl* = *fcp*.**s**-<agent constraint>.*agentSignatureSortList0* **in**
(*pl*.*length* = *sl*.*length*) ∧
(∀*i*∈1..*pl*.*length*: *isSameSort0*(*pl*[*i*].*parentAS0*.**s**-<sort>, *sl*[*i*])) **endlet**)) **endlet**)

An actual agent parameter must identify an agent definition. Its type must be a subtype of the constraint agent type (**atleast** <agent type identifier>) with no addition of formal parameters to those of the constraint type, or it must be the type denoted by <agent type identifier>, or it must be compatible with the formal <agent signature>.

An agent definition is compatible with the formal <agent signature> if the formal parameters of the agent definition have the same sorts as the corresponding <sort>s of the <sort list> of the <agent signature>, and both definitions have the same *Agent-kind*.

F2.2.5.7.3 Procedure context parameter

Concrete syntax

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

<procedure constraint> =
<procedure><identifier> | <procedure signature>

Conditions on concrete syntax

∀*fcp*∈<procedure context parameter>:∀*acp*∈<actual context parameter>:
isContextParameterCorresponded0(*fcp*, *acp*) ⇒
(**let** *td* = *getEntityDefinition0*(*acp*, **procedure**) **in**
((*fcp*.**s**-<procedure constraint>∈<identifier>) ⇒
(**let** *td'* = *fcp.contextParameterAtleastDefinition0* **in**
isDirectSubType0(*td*, *td'*) **endlet**)) ∧
((*fcp*.**s**-<procedure constraint>∈<procedure signature>) ⇒
(**let** *fpl* = *td.procedureFormalParameterList0* **in**
let *fpl'* = *fcp*.**s**-<procedure constraint>.**s**-<formal parameter>-**seq in**
(*fpl*.*length* = *fpl'*.*length*) ∧
(∀*i*∈1..*fpl*.*length*:
(*fpl*[*i*]. *parentAS0*.*parentAS0*.**s**-<parameter kind> = *fpl'*[*i*].**s**-<parameter kind>) ∨
(*fpl*[*i*]. *parentAS0*.*parentAS0*.**s**-<parameter kind>∈{**in out**, **out**}) ⇒
isSameSort0(*fpl*[*i*]. *parentAS0*.**s**-<sort>, *fpl*[*i*].**s**-<sort>))
endlet) ∧
(**let** *result* = *td*.**s**-<procedure heading>.**s**-<procedure result> **in**
let *result'*= *fcp*.**s**-<procedure constraint>.**s**-<result> **in**
((*result* = *undefined*) ∧ (*result'*= *undefined*)) ∨
((*result*≠ *undefined*) ∧ (*result'* ≠ *undefined*)∧*isSameResult0*(*result*, *result'*))
endlet))
endlet)

An actual procedure parameter must identify a procedure definition that is either a specialization of the procedure of the constraint (**atleast** <procedure identifier>) or is compatible with the formal procedure signature.

A procedure definition is compatible with the formal procedure signature if:

a) the formal parameters of the procedure definition have the same sorts as the corresponding parameters of the signature, if they have the same <parameter kind>, and if both have a result of the same <sort> or if neither returns a result; or

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

F2.2.5.7.4 Remote procedure context parameter

Concrete syntax

<remote procedure context parameter>::
<remote procedure<name> <procedure signature>

Conditions on concrete syntax

∀*fcp*∈<remote procedure context parameter>: ∀*acp*∈<actual context parameter>:
isContextParameterCorresponded0(*fcp*,*acp*) ⇒
(**let** *ps* = *getEntityDefinition0*(*acp*, **remote procedure**).**s**-<procedure signature> **in**
let *ps'*= *fcp*.**s**-<procedure signature> **in**
isSameProcedureSignature0(*ps*, *ps'*) **endlet**)

An actual parameter to a **remote** procedure context parameter must identify a <remote procedure definition> with the same signature.

Auxiliary functions

Determine if two <procedure signature>s are the same.

isSameProcedureSignature0(*ps*, *ps'*: <procedure signature>): *BOOLEAN* =def
let *fpl* = *ps.procedureSignatureParameterList0* **in**
let *fpl'* = *ps'.procedureSignatureParameterList0* **in**
(*fpl*. *length* = *fpl'.length*) ∧
(∀*i*∈1..*fpl.length*:
isSameSort0(*fpl*[*i*].**s**-<sort>, *fpl'*[*i*].**s**-<sort>) ∧
(*fpl*[*i*].**s**-<parameter kind> = *fpl'*[*i*].**s**-<parameter kind>))∧
isSameResult0(*ps*.**s**-<result>, *ps'*.**s**-<result>))
endlet

F2.2.5.7.5 Signal context parameter

Concrete syntax

<signal context parameter list> :: <signal context parameter name>+

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

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

<signal signature> = <sort list>

Conditions on concrete syntax

∀*fcp*∈<signal context parameter name>: ∀*acp*∈<actual context parameter>:
isContextParameterCorresponded0(*fcp*, *acp*) ⇒
(**let** *sd* = *getEntityDefinition0*(*acp*, **signal**) **in**
((*fcp*.**s**-<signal constraint>∈<identifier>) ⇒
(**let** *sd'* =*fcp.contextParameterAtleastDefinition0* **in**
isSubtype0(*sd*, *sd'*) **endlet**)) ∧
((*fcp*.**s**-<signal constraint>∈<signal signature>) ⇒

isSameSortList0(*sd*.**s-**<sort>**-seq**, *fcp*.**s-**<signal constraint>.**s-**<sort>**-seq**))
endlet)

An actual signal parameter must identify a signal definition that is either a subtype of the signal type of the constraint (**atleast** <signal identifier>) or compatible with the formal signal signature.

F2.2.5.7.6 Variable context parameter

Concrete syntax

<variable context parameter list> :: <variable context parameter names>+

<variable context parameter names> :: <variable<name>+ <variable constraint>

<variable constraint> = <sort>

Conditions on concrete syntax

∀*fcp*∈<name>:∀*acp*∈<actual context parameter>:
(*fcp*.*parentAS0*∈<variable context parameter names>) ∧
(*fcp*.*parentAS0*.*parentAS0*∈<variable context parameter list>) ∧
isContextParameterCorresponded0(*fcp*, *acp*) ⇒
(**let** *vd* = *getEntityDefinition0*(*acp*, **variable**) **in**
isSameSort0(*fcp*.*parentAS0*.**s-**<sort>, *vd*.**s-**<sort>)
endlet)

An actual parameter must be a variable or a formal agent or procedure parameter of the same sort as the sort of the constraint.

F2.2.5.7.7 Remote variable context parameter

Concrete syntax

<remotevariable context parameter list> ::
<remotevariable contextparameter names>+

<remotevariable contextparameter names> ::
<remote variable<name>+ <variable constraint>

Conditions on concrete syntax

∀*fcp*∈<name>:∀*acp*∈<actual context parameter>:
(*fcp*.*parentAS0*∈<remotevariable contextparameter names>) ∧
(*fcp*.*parentAS0*.*parentAS0*∈<remotevariable context parameter list>) ∧
isContextParameterCorresponded0(*fcp*, *acp*) ⇒
(**let** *vd* = *getEntityDefinition0*(*acp*, **remote variable**) **in**
isSameSort0(*fcp*.*parentAS0*.**s-**<sort>, *vd*.**s-**<sort>)
endlet)

An actual parameter must identify a <remote variable definition> of the same sort.

F2.2.5.7.8 Timer context parameter

Concrete syntax

<timer context parameter list> :: <timer context parameter name>+

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

<timer constraint> = <sort list>

Conditions on concrete syntax

∀*fcp*∈<timer context parameter name>: ∀*acp*∈<actual context parameter>:
isContextParameterCorresponded0(*fcp*, *acp*) ⇒
(**let** *td* = *getEntityDefinition0*(*acp*, **timer**) **in**
(*fcp*.**s-**<timer constraint> ≠*undefined*) ⇒
isSameSortList0(*fcp*.**s-**<timer constraint>.**s-**<sort>**-seq**, *td*.**s-**<sort>**-seq**)
endlet)

An actual timer parameter must identify a timer definition that is compatible with the formal sort constraint list. A timer definition is compatible with a formal sort constraint list if the sorts of the timer are the same sorts as in the sort constraint list.

F2.2.5.7.9 Synonym context parameter

Concrete syntax

<synonym context parameter list> ::
    <synonym context parameter name>+

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

<synonym constraint> = <sort>

Conditions on concrete syntax

∀*fcp*∈<synonym context parameter name>: ∀*acp*∈<actual context parameter>:
  isContextParameterCorresponded0(*fcp*, *acp*) ⇒
    (**let** *sd* = *getEntityDefinition0*(*acp*, **synonym**) **in**
      isSameSort0(*sd*.**s**-<sort>, *fcp*.**s**-<synonym constraint>.**s**-<sort>)
    endlet)

An actual synonym must be a constant expression of the same sort as the sort of the constraint.

F2.2.5.7.10 Sort context parameter

Concrete syntax

<sort context parameter> :: <sort<name> [<sort constraint>]

<sort constraint> = <sort> | <sort signature>

<sort signature> =
  <literal signature>* <operation signature>* <operation signature>*

Conditions on concrete syntax

∀*fcp*∈<sort context parameter>:∀*acp*∈<actual context parameter>:
  isContextParameterCorresponded0(*fcp*, *acp*) ∧ (*fcp*.**s**-<sort constraint>≠*undefined*)⇒
    (**let** *td* = *getEntityDefinition0*(*acp*, **type**).*derivedDataType0* **in**
      ((*fcp*.**s**-<sort constraint>∈<sort>)⇒
        (**let** *td'* = *fcp.contextParameterAtleastDefinition0* **in**
          (*td*.**s**-<data type specialization>.**s**-<renaming>= *undefined*) ∧
          isSubtype0(*td*, *td'*) **endlet**)) ∧
      ((*fcp*.**s**-<sort constraint>∈<sort signature>)⇒
        (∀*ls*∈<literal signature>: (*ls*.*parentAS0*= *fcp*.**s**-<sort constraint>)⇒
          ∃*ls'*∈<literal signature>:
            (*ls'*.*surroundingScopeUnit0*= *td*) ∧ *isSameLiteralSignature0*(*ls*,*ls'*)) ∧
            (∀*os*∈<operation signature>:
              (*os*.*parentAS0*= *fcp*.**s**-<sort constraint>)⇒
                ∃*os'*∈<operation signature>:
                  (*os'*.*parentAS0*∈<operator list> ∪ <method list>) ∧
                  (*os'*.*surroundingScopeUnit0*=*td*) ∧ *isSameOperationSignature0*(*os*, *os'*)))
    endlet)

An actual sort must be either a subtype without <renaming> of the sort of the constraint (**atleast** <sort>), or compatible with the formal sort signature. A sort is compatible with the formal sort signature if the literals of the sort include the literals in the formal sort signature and the operations defined by the data type that introduced the sort include the operations in the formal sort signature and the operations have the same signatures.

∀*ls*∈<literal signature>:
  (*ls*.*parentAS0*∈<sort signature>) ∧ (*ls*.*parentAS0*.*parentAS0*∈<sort context parameter>) ⇒
    (*ls*∉<named number>)

The <literal signature> must not contain <named number>.

Auxiliary functions

Get the data type definition from which a syntype definition is derived.

derivedDataType0(*sd*: <syntype definition>∪<data type definition>):<data type definition> =$_{def}$
if (*sd*∈<syntype definition>) **then** *sd*.*parentDataType0*
else *sd*
endif

Get the parent data type definition of a syntype definition.

parentDataType0(*sd*: <syntype definition>):<data type definition> =$_{def}$
if (*sd*.**s**-<parent sort identifier> = *undefined*) **then** *sd*
else
let *pd* = *getEntityDefinition0*(*sd*.**s**-<parent sort identifier>, **type**) **in**
if (*pd*∈<data type definition>) **then** *pd*
else *pd*.*parentDataType0*
endif
endlet
endif

Determine if two <literal signature>s are the same.

isSameLiteralSignature0(*ls*: <literal signature>, *ls'*: <literal signature>): *BOOLEAN*=$_{def}$
((*ls*∈<literal name>) ∧ (*ls'*∈<literal name>) ⇒ (*ls* = *ls'*)) ∧
((*ls*∈<named number>) ∧ (*ls'*∈<named number>) ⇒
(*ls*.**s**-<literal name> = *ls'*.**s**-<literal name>) ∧
(*ls*.**s**-<simple expression>.*value0*= *ls'*.**s**-<simple expression>.*value0*))

Determine if two <operation signature>s are the same.

isSameOperationSignature0(*os*: <operation signature>, *os'*: <operation signature>): *BOOLEAN*=$_{def}$
(*os.virtuality0 = os'. virtuality0*) ∧
(*os.visibility0 = os'.visibility0*) ∧
(*os*.**s-implicit**∈<operation name>⇒*os*.**s-implicit** = *os'*.**s-implicit**) ∧
(**let** *fpl* = *os.operationSignatureParameterList0*,
fpl' = *os'.operationSignatureParameterList0* **in**
(*fpl.length* = *fpl'.length*) ∧
(∀*i*∈1..*fpl.length*:
(*fpl*[*i*].**s**-<formal parameter>.**s**-<parameter kind> =
fpl'[*i*].**s**-<formal parameter>.**s**-<parameter kind>) ∧
isSameSort0(*fpl*[*i*].**s**-<formal parameter>.**s**-<sort>, *fpl'*[*i*].**s**-<formal parameter>.**s**-<sort>)) ∧
isSameResult0(*os*.**s**-<result>, *os'*.**s**-<result>)
endlet)

F2.2.5.7.11 Composite state context parameter

Concrete syntax

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

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

<composite state type signature> = <sort list>

Conditions on concrete syntax

∀*fcp*∈<compositestate type context parameter>: ∀*acp*∈<actual context parameter>:
isContextParameterCorresponded0(*fcp*, *acp*)∧
(*fcp*.**s**-<composite state type constraint>≠ *undefined*) ⇒
(**let** *td* = *getEntityDefinition0*(*acp*, **state type**) **in**

((*fcp*.**s-**<composite state type constraint>∈<identifier>) ⇒
(**let** *td' = fcp.contextParameterAtleastDefinition0* **in**
(*td.agentLocalFormalParameterList0 = empty*) ∧ *isSubtype0*(*td*, *td'*) **endlet**)) ∧
((*fcp*.**s-**<composite state type constraint>∈<composite state type signature>) ⇒
(**let** *sl = fcp*.**s-**<composite state type constraint> **in**
let *pl = td.agentFormalParameterList0* **in**
(*sl.length = pl.length*) ∧
(∀*i*∈1..*sl.length*: *isSameSort0*(*sl*[*i*], *pl*[*i*].*parentAS0*.**s-**<sort>))
endlet))
endlet)

An actual composite state type parameter must identify a composite state type definition. Its type must be a subtype of the constraint composite state type (**atleast** <composite state type identifier>) with no addition of formal parameters to those of the constraint type or it must be compatible with the formal composite state type signature.

A composite state type definition is compatible with the formal composite state type signature if the formal parameters to the composite state type definition have the same sorts as the corresponding <sort>s of the <composite state type constraint>.

F2.2.5.7.12 Gate context parameter

Concrete syntax

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

Conditions on concrete syntax

∀*fcp*∈<gate context parameter>:∀*acp*∈<actual context parameter>:
isContextParameterCorresponded0(*fcp*, *acp*) ⇒
(**let** *gd=getEntityDefinition0*(*acp*, **gate**) **in**
(*gd*.**s-**<gate> = *fcp*.**s-**<gate>) ∧
(∀*gc*∈<gate constraint>:∀*gc'*∈<gate constraint>:
(*gc.parentAS0 = gd*) ∧ (*gc'.parentAS0= fcp*) ∧
((*gc.direction0*=**out**) ∧ (*gc'. direction0*=**out**) ⇒
gc'.**s-**<sort>**-seq** *.signalSet0*) ⊆*gc*.**s-**<sort>**-seq** *.signalSet0*) ∧
((*gc. direction0*=**in**) ∧ (*gc'. direction0*=**in**) ⇒
gc.**s-**<sort>**-seq** *.signalSet0*⊆*gc'*.**s-**<sort>**-seq** *.signalSet0*)
endlet)

An actual gate parameter must identify a gate definition. Its outward gate constraint must contain all elements mentioned in the <signal list> of the corresponding formal gate context parameter. The inward gate constraint of the formal gate context parameter must contain all elements in the <signal list> of the actual gate parameter.

F2.2.5.7.13 Interface context parameter

Concrete syntax

<interface context parameter list> = <interface context parameter name>+

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

<interface constraint> :: <interface<identifier>

Conditions on concrete syntax

∀*fcp*∈<interface context parameter name>: ∀*acp*∈<actual context parameter>:
isContextParameterCorresponded0(*fcp*, *acp*) ∧ (*fcp*.**s-**<interface constraint>≠ *undefined*) ⇒
(∃*td*∈<interface definition>:
(*td =getEntityDefinition0*(*acp*.**s-**<identifier>, **interface**)) ∧
isSubtype0(*td*, *fcp.contextParameterAtleastDefinition0*))

An actual interface parameter must identify an interface definition. The type of the interface must be a subtype of the interface type of the constraint (**atleast** <interface identifier>).

F2.2.5.8 Specialization

F2.2.5.8.1 Adding properties

Concrete syntax

<specialization> :: <type expression>

Mapping to abstract syntax

| <specialization>(*x*) => *Mapping*(*x*)

Auxiliary functions

The function *specialization0* is used to get the <specialization> part of an entity definition.

specialization0(*def*: *ENTITYDEFINITION0*): <specialization> =$_{def}$
take({ *s*∈<specialization>∪<data type specialization>∪<interface specialization>:
s.surroundingScopeUnit0 = def})

F2.2.5.8.2 Virtual type

Concrete syntax

<virtuality> = **virtual | redefined | finalized**

<virtuality constraint> :: <identifier>

Conditions on concrete syntax

∀*td*∈<block type definition>∪<process type definition>∪<procedure definition>∪
<composite state type definition>:
isVirtualType0(*td*) ⇒ *td.virtualTypeAtleastDefinition0.entityKind0 = td.entityKind0*

Every virtual type has associated a virtuality constraint which is an <identifier> of the same entity kind as the virtual type.

∀*td*∈<block type definition>∪<process type definition>∪<procedure definition>∪
<composite state type definition>:
td.isVirtualType0 ⇒
¬(*td.isParameterizedType0*) ∧ ¬(*isParameterizedType0*(*td.virtualTypeAtleastDefinition0*))

A virtual type and its constraints cannot have context parameters.

∀*vc*∈<virtuality constraint>: *isVirtualType0*(*vc.surroundingScopeUnit0*)

Only virtual types may have <virtuality constraint> specified.

∀*r*∈<block type reference> ∪<process type reference> ∪<composite state type reference> ∪
<procedure reference>:
∀*d*∈<block type definition>∪<process type definition>∪<composite state type definition>∪
<procedure definition>:
(*r.referencedDefinition0 = d*)∧(*r.virtuality0* ≠ *undefined*)∧(*d.virtuality0* ≠ *undefined*) ⇒
(*r.virtuality0 = d.virtuality0*)

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

∀*td*∈<block type definition>∪<process type definition>: *isVirtualType0*(*td*) ⇒
(**let** *td' = td.virtualTypeAtleastDefinition0* **in**
(**let** *fpl = td.agentFormalParameterList0***in**
let *fpl' = td'.agentFormalParameterList0* **in**
isSameAgentFormalParameterList0(*fpl*, *fpl'*) **endlet**) ∧

(∀*gd'*∈<gate in definition>: *gd'*∈ *td'.gateDefinitionSet0* ⇒
∃*gd*∈<gate in definition>: (*gd*∈ *td.gateDefinitionSet0*) ∧ *isSameGate0*(*gd*, *gd'*)) ∧
(∀*fd'*∈<interface definition>: *fd'*∈ *td'.interfaceDefinitionSet0* ⇒
∃*fd*∈<interface definition>: (*fd*∈ *td.interfaceDefinitionSet0*) ∧ *isSameInterface0*(*fd*, *fd'*)) ∧
(∀*ed*∈*ENTITYDEFINITION0*: *ed.surroundingScopeUnit0 = td*⇒
∃*ed'*∈*ENTITYDEFINITION0*: (*ed'.surroundingScopeUnit0 = td'*) ∧ (*ed*= *ed'*)) **endlet**)

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

∀*td*∈<composite state type definition>: *isVirtualType0*(*td*) ⇒
(**let** *td'* = *td.virtualTypeAtleastDefinition0* **in**
∀*scp'*∈<state connection points>: *scp'*∈*td'.stateConnectionPointSet0* ⇒
∃*scp*∈<state connection points>:
(*scp*∈*td.stateConnectionPointSet0*) ∧ *isSameStateConnectionPoint0*(*scp*, *scp'*) **endlet**)

A virtual state type must have at least the same state connection points as its constraint.

∀*td*∈<procedure definition>: *isVirtualType0*(*td*) ⇒
(**let** *fpl* = *td.procedureFormalParameterList0* **in**
let *fpl'* = *td.virtualTypeAtleastDefinition0.procedureFormalParameterList0* **in**
isSameProcedureFormalParameterList0(*fpl*, *fpl'*) **endlet**)

A virtual procedure must have exactly the same formal parameters as its constraint.

∀*td*∈<block type definition>∪<process type definition>∪<procedure definition>∪
<composite state type definition>:
isVirtualType0(*td*) ⇒
(∀*sp*∈<specialization>:∀*vc*∈<virtuality constraint>:
(*sp.surroundingScopeUnit0 = td*) ∧ (*vc.surroundingScopeUnit0 = td*) ⇒
((*td.virtualTypeInheritsDefinition0= td.virtualTypeAtleastDefinition0*) ∨
isSubtype0(*td.virtualTypeInheritsDefinition0*, *td.virtualTypeAtleastDefinition0*)))

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

∀*td*, *td'*∈<block type definition>∪<process type definition>∪<procedure definition>∪
<composite state type definition>:
isRedefinedType0(*td*) ∧ (*td'=td.superCounterpart0*) ⇒
(**let** *sp=td.specialization0***in**
let *vc* = *take*({*vc*∈<virtuality constraint>: *vc.surroundingScopeUnit0 = td'*}) **in**
(*sp*≠*undefined*) ∧ (*vc=undefined*) ⇒
isSubtype0(*td.virtualTypeInheritsDefinition0*, *td'.virtualTypeAtleastDefinition0*) **endlet**)

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

Mapping to abstract syntax

The <virtuality constraint> is always ignored in the mapping.

Auxiliary functions

Determine if a <block type definition>, a <process type definition>, an <internal procedure definition> or a <composite state type definition> is a virtual type.

isVirtualType0(*td*: <block type definition>∪<process type definition>∪<internal procedure definition>∪
<composite state type definition>): *BOOLEAN*=$_{\text{def}}$
td.virtuality0∈{**virtual**, **redefined**}

Determine if a <block type definition>, a <process type definition>, an <internal procedure definition> or a <composite state type definition> is a redefined type.

isRedefinedType0(*td*: <block type definition>∪<process type definition>∪<internal procedure definition>
∪<composite state type definition>): *BOOLEAN*=$_{\text{def}}$

td.virtuality0∈{**redefined**, **finalized**}

Get the virtuality for a definition.

virtuality0(*td*: *DefinitionAS0*): {**virtual**, **redefined**, **finalized**} =$_{def}$
case *d* **of**
| <block type definition>=>
d.**s**-<block type heading>.**s**-<type preamble>.**s**-<virtuality>
| <process type definition> =>
d.**s**-<process type heading>.**s**-<type preamble>.**s**-<virtuality>
| <composite state type definition> =>
d.**s**-<composite state type heading>.**s**-<virtuality>
| <internal procedure definition> =>
d.**s**-<procedure heading>.**s**-<procedure preamble>.**s**-<type preamble>.**s**-<virtuality>
| <operation definition> =>
d.**s**-<operation heading> .**s**-<operation preamble>.**s**-<virtuality>
| <interface definition> => *d*.**s**-<virtuality>
| <block type reference> ∪<process type reference> ∪<composite state type reference> ∪
<procedure reference> => *d*.**s**-<type preamble>.**s**-<virtuality>
| <operation signature> => *d*.**s**-<operation preamble>.**s**-<virtuality>
| <start>∪<input part>∪<priority input>∪<save part>∪<spontaneous transition>∪
<continuous signal>∪<connect part>∪<default initialization> => *d*.**s**-<virtuality>
| <statements> => *d.parentAS0*.**s**-<virtuality>
otherwise *undefined*
endcase

Get the entity definition referred by a <virtuality constraint>.

virtualTypeAtleastDefinition0(*td*: <block type definition>∪<process type definition>∪
<procedure definition>∪<composite state type definition>): <block type definition>∪
<process type definition>∪<procedure definition>∪<composite state type definition> =$_{def}$
let *vc*=*take*({*vc*∈<virtuality constraint>: *vc.surroundingScopeUnit0* = *td*}) **in**
if *vc* ≠ *undefined* **then** *getEntityDefinition0*(*vc*, *td.entityKind0*) **else** *td* **endif**
endlet

Determine if two agent formal parameter lists are the same.

isSameAgentFormalParameterList0(*fpl*: <name>*, *fpl'*:<name>*): *BOOLEAN*=$_{def}$
(*fpl.length* = *fpl'.length*) ∧
(∀*i*∈1..*fpl.length*:(*fpl*[*i*]=*fpl'*[*i*])∧*isSameSort0*(*fpl*[*i*].*parentAS0*.**s**-<sort>,*fpl'*[*i*].*parentAS0*.**s**-<sort>))

Determine if two <gate in definition>s are the same.

isSameGate0(*gd*: <gate in definition>, *gd'*: <gate in definition>): *BOOLEAN*=$_{def}$
if (*gd*∈<textual gate definition>)∧(*gd'*∈<textual gate definition>) **then**
(*gd*.**s**-<gate>=*gd'*.**s**-<gate>)∧
(∀*gc*∈ *gd*.**s**-<gate constraint>:∃*gc'*∈ *gd'*.**s**-<gate constraint>:
(*gc.direction0* = *gc'.direction0*)∧
(*gc*.**s**-<textual endpoint constraint> =*gc'*.**s**-<textual endpoint constraint>)∧
isSameSortList0(*gc*.**s**-<sort>-**seq**, *gc'*.**s**-<sort>-**seq**))∧
(∀*gc'*∈ *gd'*.**s**-<gate constraint>:∃*gc*∈ *gd*.**s**-<gate constraint>:
(*gc.direction0* = *gc'.direction0*)∧
(*gc*.**s**-<textual endpoint constraint> =*gc'*.**s**-<textual endpoint constraint>)∧
isSameSortList0(*gc*.**s**-<sort>-**seq**, *gc'*.**s**-<sort>-**seq**))
else if (*gd*∈<textual interface gate definition>)∧(*gd'*∈<textual interface gate definition>) **then**
gd.**s**-<identifier>=*gd'*.**s**-<identifier>
else *false*
endif

Determine if two <interface definition>s are the same.

isSameInterface0(*id*:<interface definition>, *id'*:<interface definition>): *BOOLEAN*=$_{def}$
(*id.virtuality0* = *id'.virtuality0*)∧

(*id.entityName0 = id'.entityName0*)∧
(*id.entityDefinitionSet0= id'.entityDefinitionSet0*)

Get all the entity definitions defined in a scope unit.

entityDefinitionSet0(*su*: *SCOPEUNIT0*): *ENTITYDEFINITION0* =$_{def}$
{*ed*∈*ENTITYDEFINITION0*: *isDefinedIn0*(*ed, su*)}

Get all the interface definitions defined in a scope unit.

interfaceDefinitionSet0(*d*: *SCOPEUNIT0*):<interface definition>-**set** =$_{def}$
{*fd*∈<interface definition>: *isDefinedIn0*(*fd, d*)}

Get the set of <state connection points> defined in a <composite state type definition> or a <composite state>.

stateConnectionPointSet0(*td*:<composite state type definition>∪<composite state>):
<state connection points>-**set** =$_{def}$
{*scp*∈<state connection points>:
(*scp.parentAS0=td*)∧
(∃*td'*∈<composite state type definition>: *isSubtype0*(*td, td'*)∧ (*scp.parentAS0=td'*))}

Determine if two <state connection points>s are the same.

isSameStateConnectionPoint0(*scp*: <state connection points>, *scp'*: <state connection points>):
BOOLEAN=$_{def}$
{*n*∈<name>: *isAncestorAS0*(*scp,n*)} = {*n'*∈<name>: *isAncestorAS0*(*scp',n*)}

Determine if two procedure formal parameter lists are the same.

isSameProcedureFormalParameterList0(*fpl*: <name>*, *fpl'*: <name>*): *BOOLEAN*=$_{def}$
(*fpl.length = fpl'.length*) ∧
(∀*i*∈1..*fpl.length*:
(*fpl*[*i*].*parentAS0.parentAS0*.**s**-<parameter kind> =
fpl'[*i*].*parentAS0.parentAS0*.**s**-<parameter kind>) ∧
(*fpl*[*i*] = *fpl'*[*i*]) ∧ *isSameSort0*(*fpl*[*i*].*parentAS0*.**s**-<sort>, *fpl'*[*i*].*parentAS0*.**s**-<sort>))

Get the entity definition specialized by a virtual type.

virtualTypeInheritsDefinition0(*td*: <block type definition>∪<process type definition>∪
<procedure definition>∪<composite state type definition>): <block type definition>∪
<process type definition>∪<procedure definition>∪<composite state type definition>=$_{def}$
let *sp=td.specialization0* **in**
if (*sp* ≠ *undefined*) **then** *sp*.**s**-<type expression>.*baseType0*
else
let *vc=take*({*vc*∈<virtuality constraint>: *vc.surroundingScopeUnit0 = td*}) **in**
case *td.virtuality0* **of**
| **virtual** =>
if (*vc = undefined*) **then** *undefined* **else** *td.virtualTypeAtleastDefinition0* **endif**
| **redefined** =>
if (*vc = undefined*) **then** *td.superCounterpart0.virtualTypeAtleastDefinition0*
else *td.virtualTypeAtleastDefinition0*
endif
| **finalized** => *td.superCounterpart0.virtualTypeAtleastDefinition0*
endcase
endlet
endif
endlet

For a given entity definition, get the counterpart in the super type of the surrounding scope unit.

superCounterpart0(*td*: <block type definition>∪<process type definition>∪<procedure definition>∪
<composite state type definition>∪<operation definition>∪<operation signature>):

(<block type definition>∪<process type definition>∪<procedure definition>∪
<composite state type definition>∪<operation definition>)**-set** =def
{*td'*∈<block type definition>∪<process type definition>∪<procedure definition>∪
<composite state type definition>∪<operation definition>∪<operation signature>:
isSuperCounterpart0(*td', td*)}

Determine if an entity definition is the counterpart of the other one.

isSuperCounterpart0(*td*: *DefinitionAS0*, *td'*: <block type definition>∪<process type definition>∪
<procedure definition>∪<composite state type definition>∪
<operation definition>): *BOOLEAN* =def
(*td.name0*= *td'.name0*) ∧ (*td.kind0*= *td'.kind0*) ∧
(*td.virtuality0*∈{**virtual**, **redefined**})∧(*td'.virtuality0*∈{**redefined**, **finalized**}) ∧
isDirectSubType0(*td'.surroundingScopeUnit0, td.surroundingScopeUnit0*)

F2.2.5.8.3 Virtual transitions/save

Conditions on concrete syntax

∀*ad*∈<agent definition> ∪<textual typebased agent definition> ∪<composite state> ∪
<typebased composite state>:
(∀*s*∈<start>: *s*∈*ad.startSet0*⇒*s.virtuality0 = undefined*) ∧
(∀*s*∈<state>: (*s*∈*ad.stateSet0*)⇒
(*s*.**s-**<input part>.*virtuality0 = undefined*) ∧
(*s*.**s-**<priority input>.*virtuality0 = undefined*) ∧
(*s*.**s-**<save part>.*virtuality0 = undefined*) ∧
(*s*.**s-**<spontaneous transition>.*virtuality0 = undefined*) ∧
(*s*.**s-**<continuous signal>.*virtuality0 = undefined*))

Virtual transitions or saves must not appear in agent (set of instances) definitions, or in composite state definitions.

∀*s*∈<state>: |{*st*∈<spontaneous transition>: (*st.parentAS0*=*s*) ∧ (*st.virtuality0*≠*undefined*)}| ≤ 1

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

(∀*ip*∈<input part>:
(*ip.virtuality0*≠ *undefined*) ⇒ *ip*.**s-**<input list>∈<asterisk input list>) ∧
(∀*sp*∈<save part>:
(*sp.virtuality0* ≠ *undefined*) ⇒ *sp*.**s-**<save item>∈<asterisk save list>)

An input or save with <virtuality> must not contain <asterisk>.

Auxiliary functions

Get the set of <start> defined in a given definition.

startSet0(*td*: <agent definition> ∪<textual typebased agent definition> ∪<agent type definition> ∪
<composite state> ∪ <typebased composite state>∪<composite state type definition>∪
<textual typebased state partition def> ∪<state partition>∪
<internal procedure definition>):<start>**-set** =def
case *td* **of**
| <agent definition>∪
<agent type definition>∪
<composite state type definition>∪
<internal procedure definition> =>
td.localStartSet0∪*td.inheritedStartSet0*
| <textual typebased agent definition> ∪ <textual typebased state partition def> =>
let *te*=*take*({*te*∈<type expression>: *te.parentAS0.parentAS0* = *td*}) **in**
te.baseType0.startSet0
endlet
| <composite state> =>
if *td*.**s-**<composite state structure>.**s-implicit** ∈<composite state body> **then**
{*s*∈<start>: *s.parentAS0*=*td*.**s-**<composite state structure>.**s-implicit**}

else {*s*∈*sp.startSet0*:
sp∈<state partition>∧*sp.parentAS0*= *td*.**s-**<composite state structure>.**s-implicit**}
endif
| <typebased composite state> => *td*.**s-**<type expression>.*baseType0.startSet0*
| <composite state reference> => *td.referencedDefinition0.startSet0*
otherwise ∅
endcase

localStartSet0(*td*: <agent definition> ∪<agent type definition> ∪<composite state type definition>∪
<internal procedure definition>):<start>-**set** =$_{def}$
case *td* **of**
| <agent definition> ∪ <agent type definition> =>
if *td*.**s-**<agent structure>.**s-implicit** ∈ <agent body> **then**
{*td*.**s-**<agent structure>.**s-implicit**.**s-**<start>}
else *td*.**s-**<agent structure>.**s-implicit**.**s-**<state partition>.*startSet0*
endif
| <composite state type definition> =>
if *td*.**s-**<composite state structure>.**s-implicit**∈<composite state body> **then**
{*s*∈<start>: *s.parentAS0=td*.**s-**<composite state structure>.**s-implicit** }
else {*s*∈*sp.startSet0*:
sp∈<state partition>∧*sp.parentAS0=td*.**s-**<composite state structure>.**s-implicit**}
endif
| <internal procedure definition> =>
if *td*.**s-implicit**∈<procedure body> **then** { *td*.**s-implicit**.**s-**<start>}
else ∅
endif
otherwise ∅
endcase

inheritedStartSet0(*td*: <agent definition> ∪<agent type definition> ∪
<composite state type definition> ∪<internal procedure definition>): <start>-**set** =$_{def}$
let *sp=td.specialization0* **in**
if *sp=undefined* **then** ∅
else *sp*.**s-**<type expression>.*baseType0.startSet0*
endif
endlet

Get the set of <state> defined in a given definition.

stateSet0(*td*: <agent definition> ∪<textual typebased agent definition> ∪<agent type definition> ∪
<composite state> ∪<typebased composite state>∪<composite state type definition>∪
<textual typebased state partition def> ∪
<internal procedure definition>):<state>-**set** =$_{def}$
case *td* **of**
| <agent definition> ∪
<agent type definition> ∪
<composite state type definition>∪
<internal procedure definition> =>
td.localStateSet0∪*td.inheritedStateSet0*
| <textual typebased agent definition> ∪<textual typebased state partition def> =>
let *te* = *take*({*te* ∈ <type expression>: *te.parentAS0.parentAS0* = *td*}) **in**
te.baseType0.stateSet0
endlet
| <composite state> =>
if *td*.**s-**<composite state structure>.**s-implicit** ∈ <composite state body> **then**
{*s*∈<state>: *s.parentAS0=td*.**s-**<composite state structure>.**s-implicit**}
else {*s*∈*sp.stateSet0*:
sp∈<state partition>∧*sp.parentAS0=td*.**s-**<composite state structure>.**s-implicit**}
endif
| <typebased composite state> => *td*.**s-**<type expression>.*baseType0.stateSet0*
| <composite state reference> => *td.referencedDefinition0.stateSet0*

otherwise ∅
endcase

localStateSet0(*td*: <agent definition> ∪<agent type definition> ∪<composite state type definition>∪
<internal procedure definition>): <state>-**set** =$_{def}$
case *td* **of**
| <agent definition> ∪ <agent type definition> =>
if *td*.**s**-<agent structure>.**s-implicit** ∈ <agent body> **then**
{*s*∈<state>: *s.parentAS0* = *td*.**s**-<agent structure>.**s-implicit**}
else *td*.**s**-<agent structure>.**s**-<state machine>.**s-implicit**.*stateSet0*
endif
| <composite state type definition> =>
if *td*.**s**-<composite state structure>.**s-implicit**∈<composite state body> **then**
{*s*∈<state>: *s.parentAS0*= *td*.**s**-<composite state structure>.**s-implicit** }
else {*s*∈*sp*. *stateSet0*:
sp∈<state partition>∧*sp.parentAS0*=*td*.**s**-<composite state structure>.**s-implicit**}
endif
| <internal procedure definition> =>
if *td*.**s-implicit**∈<procedure body> **then** { *s*∈<state>: *s.parentAS0*= *td*.**s-implicit**}
else ∅
endif
otherwise ∅
endcase

inheritedStateSet0(*td*: <agent definition> ∪<agent type definition> ∪
<composite state type definition>∪<internal procedure definition>):<state>-**set** =$_{def}$
let *sp=td.specialization0* **in**
if *sp=undefined* **then** ∅
else *sp*.**s**-<type expression>.*baseType0.stateSet0*
endif
endlet

F2.2.5.8.4 Virtual method

Conditions on concrete syntax

∀*os*∈<operation signature>:
(*os.entityKind0*=**method**) ∧ (*os.virtuality0*∈{**redefined**, **finalized**}) ⇒
∃*os'*∈<operation signature>: (*os'*= *os.superCounterpart0*) ∧
isOperationSignatureCompatible0(*os*, *os'*)∧
(**let** *fpl* = *os.operationSignatureParameterList0* **in**
let *fpl'* = *os'.operationSignatureParameterList0* **in**
∀*i*∈1..*fpl.length*:
(*fpl*[*i*].**s**-<formal parameter>.**s**-<parameter kind>=
fpl'[*i*].**s**-<formal parameter>.**s**-<parameter kind>)
endlet)

When a method is redefined in a specialization, its signature must be sort compatible with the corresponding signature in the base type, and further, if the *Result* in the *Operation-signature* denotes a sort A, then the *Result* of the redefined method may only denote a sort B such that B is sort compatible to A. A redefinition of a virtual method must not change the <parameter kind> in any <argument> of the inherited <operation signature>.

Auxiliary functions

Determine if two <operation signature>s are compatible.

isOperationSignatureCompatible0(*os*: <operation signature>, *os'*: <operation signature>):
BOOLEAN=$_{def}$
isSortCompatible0(*os*.**s**-<result>.**s**-<sort>, *os'*.**s**-<result>.**s**-<sort>) ∧
(**let** *fpl* = *os.operationSignatureParameterList0* **in**
let *fpl'* = *os'.operationSignatureParameterList0* **in**

(*fpl.length = fpl'.length*) ∧
(∀*i*∈1..*fpl.length*:
(*isSortCompatible0*(*fpl*[*i*].**s**-<formal parameter>.**s**-<sort>,
fpl'[*i*].**s**-<formal parameter>.**s**-<sort>)) ∧
(*isSameSort0*(*fpl*[*i*].**s**-<formal parameter>.**s**-<sort>,
fpl'[*i*].**s**-<formal parameter>.**s**-<sort>)))
endlet)

F2.2.6 Agents

Abstract syntax

Agent-definition	::	*Agent-name* *Number-of-instances* *Agent-type-identifier*
Number-of-instances	::	*Initial-number* [*Maximum-number*] *Lower-bound*
Initial-number	=	*NAT*
Maximum-number	=	*NAT*
Lower-bound	=	*NAT*
Agent-formal-parameter	=	*Parameter*
Parameter	::	*Variable-name* *Sort-reference-identifier* *Parameter-aggregation*
Parameter-aggregation	::	*Aggregation-kind*
State-machine	::	*State-name* *Nextstate-parameters* *Composite-state-type-identifier*
State-transition-graph	::	[*State-start-node*] *Named-start-node*-**set** *State-node*-**set** *Free-action*-**set**

Conditions on abstract syntax

∀*d*∈*Agent-definition*: *d*. *agentKind1* = **system** ⇒ *d.parentAS1*∉ *Agent-type-definition*

An *Agent* with the *Agent-kind* **system** must not be contained in any other *Agent*.

∀*d* ∈ *Agent-definition*: *d*. *agentKind1* = **system** ⇒
d.**s**-*Number-of-instances*.**s**-*Initial-number* =1 ∧ *d*.**s**-*Number-of-instances*.**s**-*Maximum-number* = 1

In an *Agent* with the *Agent-kind* **system** the *Initial-number* of instances is 1 and the *Maximum-number* of instances is 1.

Concrete syntax

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

<agent structure> ::
<entity in agent>*
{ <interaction> | <agent body> }

<interaction> ::
{ <channel to channel connection>
| <channel definition>

| <agent definition>
| <block reference>
| <process reference>
| <textual typebased agent definition>}*
[<state machine>]

<state machine> =
<typebased composite state>
| <composite state list item>

<agent instantiation> ::
[<number of instances>]<agent additional heading>

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

<entity in agent> =
| <valid input signal set>
| <signal definition list>
| <signal list definition>
| <variable definition>
| <remote procedure definition>
| <remote variable definition>
| <data definition>
| <timer definition>
| <procedure reference>
| <procedure definition>
| <composite state type definition>
| <composite state type reference>
| <select definition>
| <block type definition>
| <process type definition>
| <block type reference>
| <process type reference>
| <gate in definition>

<valid input signal set> :: [<signal list>]

<number of instances> ::
[<initial number>] [<maximum number>] [<lower bound>]

<initial number> = <Natural<simple expression>

<maximum number> = <Natural<simple expression>

<lower bound> = <Natural<simple expression>

<agent formal parameters> :: { <aggregation kind> <parameters of sort> }+

<parameters of sort> :: <variable<name>+ <sort>

<agent body> ::
[<start>] {<state> | <free action>}*

Conditions on concrete syntax

$\forall$*ainst*$\in$<agent instantiation>:
ainst.**s-**<agent additional heading>.**s-**<agent formal parameters> $\neq$ *undefined*. $\Rightarrow$
ainst.**s-**< number of instances > $\neq$ *undefined*.

In <agent instantiation>, if <agent formal parameters> are present, <number of instances> shall be present, even if both <initial number> and <maximum number> are omitted.

$\forall$*in*$\in$<initial number>:$\forall$*mn*$\in$<maximum number>:
in.parentAS0 = *mn.parentAS0* $\Rightarrow$
in.**s-**<simple expression>.*value0* $\leq$ *mn*.**s-**<simple expression>.*value0* $\wedge$
mn.**s-**<simple expression>.*value0* $\geq$ 0

The <initial number> of instances must be less than or equal to <maximum number> and <maximum number> must be greater than zero.

Transformations

let *nn=newName* **in**
< <system definition>(*uses*, <system heading>(*n*, *addHead*), *body*) >
=8=> < <textual typebased system definition>(
<typebased system heading>(*n*, <type expression>(<identifier>(*undefined*, *nn*), *empty*))),
<system type definition>(*uses*,
<system type heading>(*empty*, *nn*,
<agent type additional heading>(*empty*, *undefined*, *addHead*)),
body) >

let *nn=newName* **in**
< <block definition>(*uses*, <block heading>(*, *n*, <agent instantiation>(*inst*, *addHead*)), *body*) >
=8=> < <textual typebased block definition>(
<typebased block heading>(*n*, *inst*,
<type expression>(<identifier>(*undefined*, *nn*), *empty*))),
<block type definition>(*uses*,
<block type heading>(*empty*, *undefined*, *nn*,
<agent type additional heading>(*empty*, *undefined*, *addHead*)),
body) >

let *nn=newName* **in**
< <process definition>(*uses*, <process heading>(*, *n*, <agent instantiation>(*inst*, *addHead*)), *body*) >
=8=> < <textual typebased process definition>((*n*, *inst*,
<type expression>(<identifier>(*undefined*, *nn*), *empty*)),
<process type definition>(*uses*,
<process type heading>(*empty*, *undefined*, *nn*,
<agent type additional heading>(*empty*, *undefined*, *addHead*)),
body) >

An Agent-definition has an implied anonymous agent type that defines the properties of the agent.

The following transformation is covered by the transformation for agent types.

An agent with an <agent body> or an <agent body area> is shorthand for an agent having only a state machine, but no contained agents. This state machine is obtained by replacing the <agent body> or <agent body area> by a composite state definition. This composite state definition has the same name as the agent and its State-transition-graph is represented by the <agent body> or the <agent body area>.

The following transformation is covered by the dynamic semantics.

In all agent instances, four anonymous variables of the pid sort of the agent (for agents not based on an agent type) or the pid sort of the agent type (for type based agents) are declared and are, in the following, referred to by self, parent, offspring and sender. They give a result for:

a) the agent instance (self);

b) the creating agent instance (parent);

c) the most recent agent instance created by the agent instance (offspring);

d) the agent instance from which the last input signal has been consumed (sender) (see also clause 11.3 of [ITU-T Z.101]).

These anonymous variables are accessed using pid expressions as further explained in clause 12.3.4.2 of [ITU-T Z.101].

For all agent instances present at system initialization, parent is initialized to Null.

For all newly created agent instances, sender and offspring are initialized to Null.

< <parameters of sort>(< *p* > ⌒ *rest*, *s*) > **provided** *rest* ≠ *empty* =1=>
    < <parameters of sort>(< *p* >, *s*), <parameters of sort>(*rest*, *s*) >

If many parameters are declared within one parameter declaration, this is shorthand for a list of parameter declarations.

Mapping

| <number of instances>(*init*, *max*, *lower*) =>
    mk-*Number-of-instances*(*Mapping*(*init*), *Mapping*(*max*), *Mapping*(*lower*))

| <agent formal parameters>(*param*) => *Mapping*(*param*)

| <parameters of sort>(< *name* >, *s*) => **mk-***Parameter*(*Mapping*(*name*), *Mapping*(*s*))

Auxiliary functions

The function *value0* computes the Natural value for a simple expression from the *Literal-signature* returned by *value1*. The Boolean values do not have any defined *Literal-natural* values, so true is treated as 0 and false is treated as 1.

value0(*e*:<simple expression>): *NAT*=$_{\text{def}}$
    let *booleanId* =
        mk-*Identifier* (< **mk-***Package-qualifier* (**mk-***Name* ("Predefined"))>, **mk-***Name* ("Boolean")) **in**
        let *booleanSort* = *getEntityDefinition1*(*booleanId*, *idKind1*(*booleanId*)).**s-***Sort* **in**
            if *isSuperType1*(*booleanSort*, *value1* ((*e*)).**s-***Result*.**s-***Sort-reference-identifier*)
            then
                if *value1* ((*e*)).**s-***Literal-name* = **mk-***Name*("true") **then** 0 **else** 1 **endif**
            else
                value1 ((*e*)).**s-***Literal-natural*
            endif
        endlet
    endlet

The function *simpleMapping* generates a *Constant-expression* for a <simple expression>. It assumes that transformations of infix operators into operator applications have taken place, and makes use of the restriction that only predefined names can be used in the <simple expression>.

simpleMapping(*e*:<simple expression>): *Constant-expression* =$_{\text{def}}$
    case e **of**
    | <operand5>(*undefined*, <operator application>(*ident*, *params*)) =>
        mk-*Operation-application*(*simpleMapping*(*ident*), <*simpleMapping*(x) | *x* **in** *params*>)
    | <operand5>(*undefined*, <literal identifier>(*qual*, *name*)) =>
        mk-*Literal-identifier*(*simpleMapping* (*qual*), *simpleMapping* (*name*))
    | <operand5>(*undefined*, <conditional expression>(*e1*, *e2*, *e3*)) =>
        mk-*Conditional-expression*(*simpleMapping* (*e1*), *simpleMapping* (*e2*), *simpleMapping* (*e3*))
    else
        undefined
    endcase

F2.2.6.1 System

Concrete syntax

<system definition> ::
    <package use clause>* <system heading> <agent structure>

<system heading> :: <system<name> <agent additional heading>

Conditions on concrete syntax

∀*sd*∈<system heading>: *sd.agentLocalFormalParameterList0* = *empty*

The <agent additional heading> in a <system definition> may not include <agent formal parameters>.

Transformations

let *nn*= *newName* **in**
< *c*=<channel definition>(*n*, *d*,
<channel path>(*ep1*=<channel endpoint>(**env**, *undefined*), *ep2*, *list1*),
undefined) >
provided *c.parentAS0.parentAS0.parentAS0* ∈ <system definition> =8=>
< <textual gate definition>(*nn*),
<channel definition>(*n*, *d*,
<channel path>(<channel endpoint>(**env**, *nn*), *ep2*, *list1*), *undefined*) >

let *nn*= *newName* **in**
< *c*=<channel definition>(*n*, *d*,
<channel path>(*ep1*=<channel endpoint>(**env**, *undefined*), *ep2*, *list1*),
<channel path>(*ep2*, *ep1*, *list2*)) >
provided *c.parentAS0.parentAS0.parentAS0* ∈ <system definition> =8=>
< <textual gate definition>(*nn*),
<channel definition>(*n*, *d*,
<channel path>(<channel endpoint>(**env**, *nn*), *ep2*, *list1*),
<channel path>(*ep2*, <channel endpoint>(**env**, *nn*), *list2*)) >

let *nn*= *newName* **in**
< *c*=<channel definition>(*n*, *d*,
<channel path>(*ep1*, *ep2*=<channel endpoint>(**env**, *undefined*), *list1*),
undefined) >
provided *c.parentAS0.parentAS0.parentAS0* ∈ <system definition> =8=>
< <textual gate definition>(*nn*),
<channel definition>(*n*, *d*,
<channel path>(*ep1*, <channel endpoint>(**env**, *nn*), *list1*), *undefined*) >

let *nn*= *newName* **in**
< *c*=<channel definition>(*n*, *d*,
<channel path>(*ep1*, *ep2*=<channel endpoint>(**env**, *undefined*), *list1*),
<channel path>(*ep2*, *ep1*, *list2*)) >
provided *c.parentAS0.parentAS0.parentAS0* ∈ <system definition> =8=>
< <textual gate definition>(*nn*),
<channel definition>(*n*, *d*,
<channel path>(*ep1*, <channel endpoint>(**env**, *nn*), *list1*),
<channel path>(<channel endpoint>(**env**, *nn*), *ep1*, *list2*)) >

For each <channel definition> in a system mentioning **env**, a gate with an anonymous name is added to the Agent-definition. The channel definition is changed to mention this gate in the <channel path> directed to the system environment.

F2.2.6.2 Block

Concrete syntax

<block definition> ::
<package use clause>* <block heading> <agent structure>

<block heading> ::
<qualifier> <block<name> <agent instantiation>

Transformations

The following transformation is covered by the dynamic semantics.

A block b with a state machine and variables is modelled by keeping the block b (without the variables) and transforming the state entity and variables into a separate state machine (sm) in the block b. For each variable v in b, this state machine will have a variable v and two exported procedures set_v (with an IN parameter of the sort of v) and get_v (with a return type being the sort of v). Each assignment to v from enclosed definitions is transformed to a remote call of set_v. Each occurrence

of v in expressions in enclosed definitions is transformed to a remote call of get_v. These occurrences also apply to occurrences in procedures defined in block b, as these are transformed into procedures local to the calling agents.

A block b with only variables and/or procedures is transformed as above, with the graph of the generated state machine having just one state, where it inputs the generated set and get procedures.

The channels connected to the state machine are transformed so that they are connected to sm.

This transformation takes place after types and context parameters have been transformed.

F2.2.6.3 Process

Concrete syntax

<process definition> ::
    <package use clause>* <process heading> <agent structure>

<process heading> :: <qualifier> <process<name> <agent instantiation>

F2.2.6.4 Procedure

Abstract syntax

Procedure-definition	::	*Procedure-name* *Procedure-formal-parameter** [*Result*] [*Procedure-identifier*] *Data-type-definition*-**set** *Syntype-definition*-**set** *Variable-definition*-**set** *Composite-state-type-definition*-**set** *Procedure-definition*-**set** *Procedure-graph* [*Abstract*]
Procedure-formal-parameter	=	*In-parameter* | *Inout-parameter* | *Out-parameter*
In-parameter	::	*Parameter*
Inout-parameter	::	*Parameter*
Out-parameter	::	*Parameter*
Procedure-graph	::	[*Procedure-start-node*] *State-node*-**set** *Free-action*-**set**
Result	::	*Sort-reference-identifier* *Result-aggregation*
Result-aggregation	::	*Aggregation-kind*

Concrete syntax

<procedure definition> =
        <external procedure definition>
    | <internal procedure definition>

<internal procedure definition> ::
    <package use clause>* <procedure heading> <entity in procedure>*
    { <procedure body> | <virtuality> <statements> | [<virtuality>] <compound statement>

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

<exported> :: [<remote procedure<identifier>]

<procedure heading> ::

<procedure preamble> <qualifier> <procedure<name>
[<formal context parameters>]
[<virtuality constraint>]
[<specialization>]
<procedure formal parameters>
[<procedure result>]

<procedure formal parameters> =
<formal variable parameters>*

<entity in procedure> =
<variable definition>
| <data definition>
| <procedure reference>
| <procedure definition>
| <composite state>
| <composite state type definition>
| <composite state type reference>
| <select definition>

<procedure body> ::
[<start>] {<state> | <free action>}*

<external procedure definition> :: <procedure<name> <procedure signature>

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

<parameter aggregation> :: <aggregation kind>

<parameter kind> = [**inout** | **in** | **out**]

<procedure result> :: <result aggregation> [<variable<name>] <sort>

<result aggregation> :: <aggregation kind>

<procedure signature> :: <formal parameter>* [<result>]

Conditions on concrete syntax

$\forall pd \in$ <procedure definition>:*pd.isExported0* $\Rightarrow$
pd.formalContextParameterList0 = *empty* $\wedge$
pd.surroundingScopeUnit0 $\in$ <agent type definition> $\cup$ <agent definition>

An exported procedure cannot have formal context parameters and its enclosing scope must be an agent type or agent definition.

$\forall vd \in$ <variable definition>:
vd.surroundingScopeUnit0 $\in$ <procedure definition> $\Rightarrow \neg$ *vd.isExported0*

<variable definition> in an <internal procedure definition>, cannot contain **exported** <variable name>s

$\forall pd1, pd2 \in$ <internal procedure definition>:
(*parentAS0ofKind*(*pd1*, <agent definition> $\cup$ <agent type definition>) =
parentAS0ofKind(*pd2*, <agent definition> $\cup$ <agent type definition>) $\wedge$
pd1.isExported0 $\wedge$ *pd2.isExported0* $\wedge$ *pp1* $\neq$ *pp2*) $\Rightarrow$
(*pd1*.**s-**<procedure heading>.**s-**<procedure preamble>.**s-**<exported>.**s-**<identifier> $\neq$
pd1.**s-**<procedure heading>.**s-**<procedure preamble>.**s-**<exported>.**s-**<identifier>)

Two exported procedures in an agent cannot mention the same <remote procedure identifier>.

$\forall te \in$ <type expression>: *te.baseType0* $\notin$ <external procedure definition> $\wedge$
$\forall procCons \in$ <procedure constraint>: *procCons*.**s-**<identifier> $\neq$ *undefined* $\Rightarrow$
getEntityDefinition0(*procCons*.**s-**<identifier>, **procedure**) $\notin$
<external procedure definition>

An external procedure cannot be mentioned in a <type expression> or in a <procedure constraint>.

Transformations

<formal variable parameters>(*undefined*, *params*) =1=> <formal variable parameters>(**in**, *params*)

A formal parameter with no explicit <parameter kind> has the implicit <parameter kind> in.

<internal procedure definition>(*uses*,
h=<procedure heading>(*, *, *, *, *, *, *, *,
<procedure result>(*resName*, *resSort*), *), *entities*, *body*)
provided *resName* ≠ *undefined* ∧
replaceInSyntaxTree(<return body>(*undefined*), <return body>(*resName*), *body*) ≠ *body*
=8=>
<internal procedure definition>(*uses*, *h*, *entities*,
replaceInSyntaxTree(<return body>(*undefined*), <return body>(*resName*), *body*))

When a <variable name> is present in <procedure result>, then all <return>s or <return area>s within the procedure graph without an <expression> are replaced by a <return> or <return area>, respectively, containing <variable name> as the <expression>.

p=<internal procedure definition>(*uses*,
h=<procedure heading>(*, *, *, *, *, *, *, *, <procedure result>(*resName*, *resSort*), *),
entities, *body*)
provided *resName* ≠ *undefined* ∧
resName ∉ { *v*.**s-**<name> | *v* ∈ <variables of sort gen name>: *v.parentAS0.parentAS0.parentAS0* = *p* }
=8=>
<internal procedure definition>
(*uses*, *h*,
entities ⌒ <variable definition>
(*undefined*,
< <variables of sort>
(< <variables of sort gen name>(*resName*, *undefined*) >, *resSort*, *undefined*)
>
),
body)

A <procedure result> with <variable name> is derived syntax for a <variable definition> with <variable name> and <sort> in <variables of sort>. If there is a <variable definition> involving <variable name> no further <variable definition> is added.

The following statement is covered by the dynamic semantics.

A <procedure start area> which contains <virtuality> of **virtual** or **redefined**, a <start> of a <procedure body> which contains <virtuality> of **virtual** or **redefined**, or a <statements> in an <internal procedure definition> following <virtuality> of **virtual** or **redefined** is called a virtual procedure start. Virtual procedure start is further described in clause 8.4.3 of [ITU-T Z.102].

< <external procedure definition>(*, *) > =7=> *empty*

An external procedure definition is not considered in the dynamic semantics.

Mapping to abstract syntax

| <internal procedure definition>(*,<procedure heading>(*,*,*name*,*,*,*parent*,*parms*,*result*,*),*entities*,*body*)
=> **mk-***Procedure-definition*(*Mapping*(*name*), *Mapping*(*result*), *Mapping*(*parent*),
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Data-type-definition*) },
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Syntype-definition*) },
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Variable-definition*) },
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Composite-state-type-definition*) },
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Procedure-definition*) },
Mapping(*body*))
| <formal variable parameters>(**in**, *params*) => **mk-***In-parameter*(*Mapping*(*params*))
| <formal variable parameters>(**out**, *params*) => **mk-***Out-parameter*(*Mapping*(*params*))

| <formal variable parameters>(**inout**, *params*) => **mk-***Inout-parameter*(*Mapping*(*params*))

Auxiliary functions

Determine if a <variable definition> or an <internal procedure definition> is exported.

isExported0(*vp*: <variable definition>∪<internal procedure definition>): *BOOLEAN*=$_{\text{def}}$
case *vp* **of**
| <variable definition> =>
if *vp*.**s-exported** = *undefined* **then** *false* **else** *true* **endif**
| <internal procedure definition> =>
if *vp*.**s-**<procedure heading>.**s-**<procedure preamble>.**s-**<exported> = *undefined*
then *false*
else *true*
endif
otherwise *false*
endcase

Get the formal parameter list for an <internal procedure definition>.

procedureFormalParameterList0(*pd*: <internal procedure definition>): <name>*=$_{\text{def}}$
pd.localProcedureFormalParameterList0⌒ *pd.inheritedProcedureFormalParameterList0*

localProcedureFormalParameterList0(*pd*: <internal procedure definition>): <name>* =$_{\text{def}}$
let *fpl*=*pd*.**s-**<procedure heading>.**s-**<formal variable parameters>**-seq in**
if *fpl* = *empty* **then** *empty*
else
< *f*.**s-**<parameters of sort>.**s-**<variable<name>**-seq** | *f* **in** *fpl* >
endif
endlet

inheritedProcedureFormalParameterList0(*pd*: <internal procedure definition>): <name>* =$_{\text{def}}$
let *sp* = *pd.specialization0* **in**
if *sp* = *undefined* **then** *empty*
else *sp*.**s-**<type expression>.*baseType0.procedureFormalParameterList0*
endlet

Get the formal parameter list for a <procedure signature>.

procedureSignatureParameterList0(*ps*: <procedure signature>):<formal parameter>* =$_{\text{def}}$
ps.**s-**<formal parameter>**-seq**

Get the formal parameter list for an *Agent-type-definition*, a *Composite-state-type-definition*, or a *Procedure-definition*.

formalParameterList1
(*d*: *Agent-type-definition*∪*Composite-state-type-definition*∪*Procedure-definition*):
*Parameter**=$_{\text{def}}$
d.localFormalParameterList1⌒*d.inheritedFormalParameterList1*

localFormalParameterList1
(*d*: *Agent-type-definition*∪*Composite-state-type-definition*∪*Procedure-definition*):
*Parameter**=$_{\text{def}}$
case *d* **of**
| *Agent-type-definition* => *d*.**s-***Agent-formal-parameter***-seq**
| *Composite-state-type-definition* => *d*.**s-***Composite-state-formal-parameter***-seq**
| *Procedure-definition* => *d*.**s-***Procedure-formal-parameter***-seq**
otherwise *empty* **endif**
endcase

inheritedFormalParameterList1(*d*: *Agent-type-definition*∪*Composite-state-type-definition*∪
Procedure-definition): *Parameter**=$_{\text{def}}$

```
    let d' = take({d'∈Agent-type-definition∪Composite-state-type-definition∪Procedure-definition:
            isDirectSuperType1(d', d)}) in
        if d'≠undefined then d'. formalParameterList1
        else empty endif
    endlet
```

Determine if the sort list of *Expression*s corresponds by position to the *Sort-reference-identifier* list.

```
isActualAndFormalParameterMatched1
    (expl:[Expression]*, fpsl:Sort-reference-identifier*):BOOLEAN= def
    (expl.length = fpsl.length)∧
    (∀i∈1..expl.length:expl[i]=undefined∨isCompatibleTo1(expr[i].staticSort1, fpsl[i]))
```

Get the sort list of the formal parameters of the given definition.

```
formalParameterSortList1
    (d: Agent-type-definition∪Composite-state-type-definition∪Procedure-definition∪
    Operation-signature):Sort-reference-identifier*= def
    case d of
    | Agent-type-definition∪Composite-state-type-definition∪Procedure-definition=>
        <param.s-Sort-reference-identifier | param in d.formalParameterList1>
    | Operation-signature=> <fa.s-Argument | fa in d.s-Formal-argument-seq>
    | Signal-definition∪Timer-definition=>d.s-Sort-reference-identifier-seq
    endcase
```

Get the set of state nodes included in a type definition, state node or a state partition.

```
stateNodeSet1(d: TYPEDEFINITION1∪State-node∪State-partition): State-node-set =def
    case d of
        | TYPEDEFINITION1=>d.localStateNodeSet1∪d.inheritedStateNodeSet1
        | State-node=>
            if d.s-Composite-state-type-identifier ≠ undefined then d.baseType1.stateNodeSet1∪{d}
            else {d}
        | State-partition=> d.baseType1.stateNodeSet1
        otherwise=>∅
    endcase
```

Get the set of state nodes defined locally in a type definition.

```
localStateNodeSet1(d: TYPEDEFINITION1): State-node-set =def
    case d of
        | Agent-type-definition=>
            if d.State-machine ≠ undefined then
                d. State-machine.baseType1.stateNodeSet1
            else ∅
        | Procedure-definition=>
            {sn.stateNodeSet1: sn∈d.s-Procedure-graph.s-State-node-set}
        | Composite-state-type-definition=>
            if d.s-implicit∈Composite-state-graph then
                {sn.stateNodeSet1: sn∈d.s-implicit.s-State-transition-graph.s-State-node-set}
            else // d.s-implicit∈State-aggregation-node
                {sp.stateNodeSet1: sp∈d.s-implicit.s-State-partition-seq.toSet}
        otherwise=>∅
    endcase
```

Get the set of state nodes defined in a super type.

```
inheritedStateNodeSet1(d: TYPEDEFINITION1): State-node-set =def
    case d of
        | Agent-type-definition=>
            if d.s-Agent-type-identifier≠ undefined then
                getEntityDefinition1(d.s-Agent-type-identifier, agent type).stateNodeSet1
            else ∅
```

| *Procedure-definition*=>
if *d*.**s**-*Procedure-identifier*≠ *undefined* **then**
getEntityDefinition1(*d*.**s**-*Procedure-identifier*, **procedure**).*stateNodeSet1*
else ∅
| *Composite-state-type-definition*=>
if *d*.**s**-*Composite-state-type-identifier* ≠ *undefined* **then**
getEntityDefinition1(*d*.**s**-*Composite-state-type-identifier*, **state type**).*stateNodeSet1*
else ∅
otherwise=>∅
endcase

F2.2.7 Communication

F2.2.7.1 Channel definition

Abstract syntax

Channel-definition	::	*Channel-name* [*Encoding-rules*] [**NODELAY**] *Channel-path*-**set**
Channel-path	::	*Originating-gate* *Destination-gate* *Signal-identifier*-**set**
Originating-gate	=	*Gate-identifier*
Destination-gate	=	*Gate-identifier*

Conditions on abstract syntax

∀*c*∈*Channel-definition*: |*c*.**s**-*Channel-path*-**set**| ∈ {1, 2}

The *Channel-path*-**set** contains at least one *Channel-path* and no more than two.

∀ *c*∈*Channel-definition*: |*c*.**s**-*Channel-path*| = 2 ⇒
(∀*p*,*p'*∈*c*.**s**-*Channel-path*: *p*≠*p'* ⇒
p.**s**-*Originating-gate* = *p'*.**s**-*Destination-gate* ∧ *p'*.**s**-*Originating-gate* = *p*.**s**-*Destination-gate*)

When there are two paths, the channel is bidirectional and the *Originating-gate* of each *Channel-path* must be the same as the *Destination-gate* of the other *Channel-path*.

∀*c* ∈ *Channel-definition*: ∀*p* ∈ *Channel-path*: ∀*d* ∈ *Agent-type-definition*:
∀*g*, *g'* ∈ *Gate-definition*: (*p*.*parentAS1*= *c*) ∧ (*g*. *parentAS1*= *d*) ∧ (*g'*. *parentAS1*= *d*) ∧ (*g* ≠ *g'*) ∧
(*p*.**s**-*Originating-gate*=*g*.*identifier1*) ∧ (*p*.**s**-*Destination-gate*= *g'*.*identifier1*) ⇒
|*c*.**s**-*Channel-path*-**set**|=1

If the *Originating-gate* and the *Destination-gate* are in the same *Agent-definition*, the channel must be unidirectional (there must be only one element in the *Channel-path*-**set**).

∀*c* ∈ *Channel-definition*: ∀*p* ∈ *Channel-path*: ∃*g*, *g'* ∈ *Gate-definition*:
(*p*.*parentAS1*= *c*) ∧ (*p*.**s**-*Originating-gate*=*g*. *identifier1*) ∧ (*g*.*parentAS1*= *c*.*parentAS1*) ∧
(*p*.**s**-*Destination-gate*= *g'*.*identifier1*) ∧ (*g'*. *parentAS1*=*c*. *parentAS1*)

The *Originating-gate* or *Destination-gate* must be defined in the same scope unit in the abstract syntax in which the channel is defined.

Concrete syntax

<channel definition> ::
[<channel<name> <encoding rules>] [**nodelay**] <channel path> [<channel path>]

<channel path> :: <channel endpoint> <channel endpoint> [<signal list>]

<channel endpoint> ::
{ <identifier> | **env** | **this**}[<gate>]

<as channel> ::

<channel<identifier>

Further study is needed for the handling of <as channel>.

Conditions on concrete syntax

∀*ce*∈<channel endpoint>:
let *id* = *ce*.**s-implicit in**
let *g* = *ce*.**s-**<gate> **in**
(*id*∈<identifier>∧ *getEntityDefinition0*(*id*,**agent**)∈ <textual typebased agent definition>)⇒
(**let** *d*= *getEntityDefinition0*(*id*, **agent**) **in**
let *td* = *d*.**s-**<type expression>.*baseType0* **in**
(*ce*.**s-**<gate> ≠ *undefined* ∧
(∃*gd*∈<textual gate definition>: ∃*gc*∈<gate constraint>:
gd∈*td.localGateDefinitionSet0*∧*gc.parentAS0* =*gd*∧
gd.name0=*g*∧ *gc.direction0* =*ce.direction0* ∧
gc.**s-**<signal list item>**-seq**.*signalSet0* ⌒
ce.parentAS0.**s-**<signal list item>**-seq**.*signalSet0*≠∅))
endlet)
endlet

<gate> must be specified if <channel endpoint> denotes a connection to a <textual typebased agent definition> in which case the <gate> must be defined directly in the agent type for that agent, and the gate and the channel must have at least one common element in their signal lists in the same direction.

∀*ce*∈<channel endpoint>:
let *id* = *ce*.**s-implicit in**
let *g* = *ce*.**s-**<gate> **in**
(*id*∈<identifier>∧
getEntityDefinition0(*id*, **state**)∈ <textual typebased state partition def>)⇒
(**let** *d*= *getEntityDefinition0*(*id*, **state**) **in**
let *td*= *d*. **s-**<type expression>.*baseType0* **in**
(*ce*.**s-**<gate> ≠ *undefined* ∧
(∃*gd*∈<textual gate definition>: ∃*gc*∈<gate constraint>:
gd∈*td.localGateDefinitionSet0*∧*gc.parentAS0* =*gd*∧
gd.name0=*g*∧ *gc.direction0* =*ce.direction0* ∧
gc.**s-**<signal list item>**-seq**.*signalSet0* ⌒
ce.**s-**<signal list item>**-seq**.*signalSet0*≠∅))
endlet)
endlet

<gate> must be specified if <channel endpoint> denotes a connection to a <textual typebased state partition def> in which case the <gate> must be defined directly in the state type for that state, and the gate and the channel must have at least one common element in their signal lists in the same direction.

∀*ce*∈<channel endpoint>:
let *id* = *ce*.**s-implicit in**
let *g* = *ce*.**s-**<gate> **in**
let *su* = *ce.surroundingScopeUnit0* **in**
(*id* = **env**∧*su*∈<agent type definition>) ⇒
(∃*gd*∈<textual gate definition>: ∃*gc*∈<gate constraint>:
gd∈*su.localGateDefinitionSet0*∧*gc.parentAS0* =*gd*∧
gd.name0=*g*∧ *gc.direction0* =*ce.direction0* ∧
gc.**s-**<signal list item>**-seq**.*signalSet0* ⌒
ce.parentAS0.**s-**<signal list item>**-seq**.*signalSet0*≠∅)
endlet

<gate> must be specified if **env** is specified and the channel is defined in an agent type in which case the <gate> must be defined in this agent type respectively, and the gate and the channel must have at least one common element in their signal lists in the same direction.

Transformations

Further study is needed for inserting a choice data type corresponding to the channel signals with a unique anonymous *Sort* name in the context that the *Channel-definition* is visible.

<channel definition>(*undefined, delay, p1, p2*) =1=>
    <channel definition>(*newName, delay, p1, p2*)

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

t = <textual typebased agent definition>
provided *unconnectedGates*(*t*) = *undefined*
=9=> *t*
and
    unconnectedGates(*t*):=
    (**let** *id* = *take*({ *te*.**s-**<base type> | *te* ∈ <type expression>: *te.parentAS0.parentAS0* = *t* }) **in**
        { *g* ∈ *id.refersto0.getEntities.toSet*:
            g ∈ <gate in definition> ∧ ¬*isConnected*(*g*, *undefined*, *id.refersto0.getEntities.toSet*) }
    endlet)

t = <agent type definition>
provided *unconnectedGates*(*t*) = *undefined*
=9=> *t*
and
    unconnectedGates(*t*):=
        { *g* ∈ *t.getEntities.toSet*:
            g ∈ <gate in definition> ∧ ¬*isConnected*(*g*, *undefined*, *t.getEntities.toSet*) }

< *a1* = <textual typebased agent definition> > ⌒ *something* ⌒
    < *a2* = <textual typebased agent definition> >
provided *missingConnections*(*a1*,*a2*) ≠ ∅
=10=>
    (**let** *c* = <channel definition>(*newName*, *undefined*, *missingConnections*(*a1*,*a2*).*take*, *undefined*) **in**
        < *a1* > ⌒ *something* ⌒ < *a2* > ⌒ < *c* >
    endlet)

< *a1* = <textual typebased agent definition> > ⌒ *something* ⌒
    < *a2* = <textual typebased agent definition> >
provided *missingConnections*(*a2*,*a1*) ≠ ∅
=10=>
    (**let** *c* = <channel definition>(*newName*, *undefined*, *missingConnections*(*a2*,*a1*).*take*, *undefined*) **in**
        < *a1* > ⌒ *something* ⌒ < *a2* > ⌒ < *c* >
    endlet)

< *a* = <textual typebased agent definition> >
provided *missingConnections*(*a*,*a.parentAS0.parentAS0*) ≠ ∅
=10=>
    (**let** *c* = <channel definition>(*newName*, *undefined*,
            missingConnections(*a*,*a.parentAS0.parentAS0*).*take*, *undefined*) **in**
        < *a* > ⌒ < *c* >
    endlet)

< *a* = <textual typebased agent definition> >
provided *missingConnections*(*a.parentAS0.parentAS0*,*a*) ≠ ∅
=10=>
    (**let** *c* = <channel definition>(*newName*, *undefined*,
            missingConnections(*a.parentAS0.parentAS0*,*a*).*take*, *undefined*) **in**
        < *a* > ⌒ < *c* >

endlet)

If an agent or agent type contains explicit or implicit gates that are not connected by explicit channels, implicit channels are derived according to the following three steps:

a) Step 1: Insertion of channels between instance sets inside the agent or agent type and between the instance sets and the agent state machine.

b) Step 2: Insertion of channels from a gate on the agent or agent type to gates on instance sets inside the agent or agent type and to gates on the agent state machine.

c) Step 3: Insertion of channels from gates on instance sets inside the agent or agent type and from gates on the agent state machine to gates on the agent or agent type.

In the steps of the subclauses below, two elements S1 and S2 are matching if:

a) both denote the same interface, signal, remote procedure or remote variable; or

b) S1 denotes a signal/remote procedure/remote variable, S2 denotes an interface and S1 is an element of S2; or

c) S1 and S2 denote interfaces and interface S2 inherits interface S1.

After the introduction of all implicit channels, duplicates of elements (signals, remote procedures/variable and interfaces) occurring in a single path of an implicit channel are removed.

Step 1: Insertion of implicit channels between entities inside one agent or agent type

For each agent and agent type "ParentUnit" in the specification:

– For each instance set and agent state machine reference "FromSet" in "ParentUnit":

– For each gate "FromGate" on "FromSet" that has no channels explicitly connected to it:

  – For each element "S1" in the outgoing signal list associated with "FromGate" (where the element can be either an interface, signal, remote procedure, or remote variable):

    – For each contained instance set and agent state machine reference "ToSet" in "ParentUnit" such that "ToSet" is not the same as "FromSet":

    – For each gate "ToGate" on "ToSet" for which the "ToGate" has no explicit channels connected to it and the gate contains a matching element ´S2´;

    – If there is no channel from "FromGate" to "ToGate" then create a one-directional implicit channel from "FromGate" to "ToGate". If "ParentUnit" is a process or process type then create a channel without delay, otherwise create a channel with delay. Add "S1" to the signal list attached to the channel from "FromGate" to "ToGate".

Step 2: Insertion of implicit channels from the gates on an agent or agent type

The following is applied for insertion of channels from a gate on the agent or agent type to gates on instance sets inside the agent or agent type and from the agent or agent type to the state machine of the agent or agent type.

For each agent or agent type "ParentUnit" in the specification:

– For each gate "FromGate" on "ParentUnit" that has no channels explicitly connected to it inside the agent or agent type:

  – For each element "S1" in the incoming signal list associated with "FromGate" (where the element can be either an interface, signal, remote procedure, or remote variable):

  – For each instance set or agent state machine reference "ToSet" in "ParentUnit":

  – For each gate "ToGate" on "ToSet" for which the "ToGate" has no explicit channels connected to it and the gate contains a matching element ´S2´:

– If there is no channel from "FromGate" to "ToGate" then create a one-directional implicit channel from "FromGate" to "ToGate". If "ParentUnit" is a process or process type then create a channel without delay, otherwise create a channel with delay. Add "S1" to the signal list attached to the channel.

Step 3: Insertion of implicit channels from the gates on instance sets and from the gates on the agent state machine

The following is applied for insertion of implicit channels from the gates on instance sets within the agent or agent type to the gates on the agent or agent type:

For each agent or agent type "ParentUnit" in the specification:

– For each instance set or agent state machine reference "FromSet" in "ParentUnit":

– For each gate "FromGate" on "FromSet" that has no channels explicitly connected to it:

– For each element "S1" in the outgoing signal list associated with "FromGate" (where the element can be either an interface, signal, remote procedure, or remote variable):

– For each gate "ToGate" on "ParentUnit' that has no explicit internal channels connected to it and contains a matching element ´S2´:

– If there is no channel from "FromGate" to "ToGate" then create a one-directional implicit channel from "FromGate" to "ToGate". If "ParentUnit" is a process or process type then create a channel without delay, otherwise create a channel with delay. Add "S1" to the signal list attached to the channel from "FromGate" to "ToGate".

The following statement is modelled in the dynamic semantics.

A channel with both endpoints being gates of one <textual typebased agent definition> represents individual channels from each of the agents in this set to all agents in the set, including the originating agent. Any resulting bidirectional channel connecting an agent in the set to the agent itself is split into two unidirectional channels.

Mapping to abstract syntax

```
| <channel definition>(name, delayProperty, path1, path2)
    => mk-Channel-definition(Mapping(name), Mapping(delayProperty),
          if path2=undefined
          then { Mapping(path1) }
          else { Mapping(path1), Mapping(path2) }
          endif)

| <channel path>(endp1, endp2, with)
    => mk-Channel-path(Mapping(endp1), Mapping(endp2), Mapping(with))

| <channel endpoint>(*,gate) => Mapping(gate)
```

Auxiliary functions

```
SIGNALDIRECTION0 =def { out, in }
```

Get the direction of a <gate constraint> or a <channel endpoint>.

```
direction0(p: <gate constraint>∪<channel endpoint>): SIGNALDIRECTION0=def
    case p of
    | <channel endpoint> => if p=p.parentAS0. s-<channel endpoint> then out else in endif
    | <gate constraint> => p.s-implicit
    otherwise undefined
    endcase
```

The function *origination0* gets the originating channel endpoint of a <channel path>.

```
origination0(p: <channel path>): <channel endpoint> =def
    p.s-<channel endpoint>
```

The function *destination0* gets the destination channel endpoint of a <channel path>.

```
destination0(p: <channel path>): <channel endpoint> =def
    p.s2-<channel endpoint>
```

The function *channelEndpointReferTo*0 is used to get the entity definition that the <channel endpoint> referred to.

```
channelEndpointReferTo0(ep:<channel endpoint>): ENTITYDEFINITION0=def
    let end = ep.s-implicit in
        case end of
        | <identifier>=>getEntityDefinition0(end, end.idKind0)
        | this =>parentAS0ofKind(end, <agent definition> ∪<agent type definition> )
        | env =>undefined
        endcase
    endlet
```

The function *unconnectedGates* is used to store the gates that are not explicitly connected.

```
controlled unconnectedGates: <textual typebased agent definition> ∪ <agent type definition> →
                <gate in definition>-set
```

The function *missingConnections* is used to compute the missing implicit connections between two agents.

```
missingConnections(ag1: <textual typebased agent definition> ∪ <agent type definition>,
        ag2: <textual typebased agent definition> ∪ <agent type definition>): <channel path>-set =def
    let entities =
        if ag1 ∈ <agent type definition> then ag1.getEntities
        elseif ag2 ∈ <agent type definition> then ag2.getEntities
        else parentAS0ofKind(ag1, <agent type definition>).getEntities
        endif
    in
    let id1 =
        if ag1 ∈ <agent type definition> then env else ag1.identifier0 endif
    in
    let id2 =
        if ag2 ∈ <agent type definition> then env else ag2.identifier0 endif
    in
        U { { <channel path>(<channel endpoint>(id1, g1), <channel endpoint>(id2, g2),
                inoutSignals(g1, out) ⌒ inoutSignals(g2, in))
            | g2 ∈ ag2.unconnectedGates ∧ inoutSignals(g1, out) ⌒ inoutSignals(g2, in) ≠ ∅ ∧
                ¬isConnected(g1, g2, entities)}
        | g1 ∈ ag1.unconnectedGates }
```

The function *isConnected* is used to check whether two gates are connected.

```
isConnected(g1:<gate in definition>, g2: <gate in definition>, ent: DefinitionAS0-set): BOOLEAN =def
    let allPathes =
        U{ { e.s-<channel path> } ∪
            if e.s2-<channel path> = undefined then ∅ else { e.s2-<channel path> } endif
            | e ∈ ent.toSet: e ∈ <channel definition> }
    in
        ∃ p ∈ allPathes: g1=p.s-<channel endpoint>.s-<gate>.refersto0 ∧
            (g2=p.s2-<channel endpoint>.s-<gate>.refersto0 ∨ g2 = undefined)
    endlet
```

The function *inoutSignals* is used to compute the outwards or inwards going signals of a gate.

inoutSignals(*g*: <textual gate definition>, *kind*: { **in**, **out** }): <signal list item>**-set** =$_{def}$
U{ *g*.**s-**<signal list item>**-seq** | *g* ∈ <gate constraint>: *g*.**s-implicit** = *kind* }

F2.2.7.2 Connections

Concrete syntax

<channel to channel connection> ::
<external channel identifiers> <channel<identifier>+

<external channel identifiers> = <channel<identifier>+

Conditions on concrete syntax

∀*c1*,*c2*∈<channel to channel connection>:
(**let** *ids1*= *c1*.**s2-**<identifier>**-seq**.*toSet* **in**
let *ids2*= *c2*.**s2-**<identifier>**-seq**.*toSet* **in**
c1.surroundingScopeUnit0= c2.surroundingScopeUnit0∧ *c1* ≠ *c2*⇒ *ids1*⌒*ids2* = ∅
endlet
endlet)

No channel may be mentioned after the keyword **and** in more than one <channel to channel connection> of a given scope unit.

∀*c1*,*c2*∈<channel to channel connection>:
(**let** *ids1*= *c1*.**s-**<identifier>**-seq**.*toSet* **in**
let *ids2*= *c2*.**s-**<identifier>**-seq**.*toSet* **in**
(*c1.surroundingScopeUnit0= c2.surroundingScopeUnit0*∧ *c1* ≠ *c2*)
⇒(*ids1* = *ids2* ∨ *ids1*⌒*ids2* = ∅)
endlet
endlet)

For any pair of <channel to channel connection> items of a given scope unit, the <external channel identifiers>s shall either mention the same set of channels, or shall have no channels in common.

Transformations

let *nn*=*newName* **in**
< *c*=<channel to channel connection>(*, *) > **provided** *c.myImplicitGateIdentifier = undefined*
=8=> < *c*, <textual gate definition>(*nn*,
<gate constraint>(**out**, *allSignalsOut*(*c*)), <gate constraint>(**in**, *allSignalsIn*(*c*))) >
and
c.myImplicitGateIdentifier:= <identifier>(*fullQualifier0*(*c*), *nn*)

Each different <channel to channel connection> in a given scope unit defines one implicit gate on the scope unit. All channels in the <channel to channel connection> are connected to that gate in their respective scope units. The gate constraints of the implicit gate are derived from the channels connected to the gate.

c=<channel endpoint>(*id*, *undefined*) **provided** *findconnect*(*c.parentAS0.parentAS0*, *id*) ≠ *undefined*
=8=> <channel endpoint>(*id*, *findconnect*(*c.parentAS0.parentAS0*, *id*))

The name of the gate is a unique and unambiguous derived name. In the surrounding scope unit the <channel definition> that is identified by the <channel identifier> is extended with a **via** <gate> part. The **via** <gate> part is added to the <channel endpoint> that references the current scope unit and it mentions the implicit gate. Inside the scope unit the channels that are associated with the external channel by means of the <channel to channel connection> are modified, by extending the <channel endpoint> that mentions **env** with a **via** <gate> part for the implicit gate.

Auxiliary functions

We introduce an auxiliary function to store the implicitly generated gate identifier of a connection.

controlled *myImplicitGateIdentifier*: <channel to channel connection> → <identifier>

The function *findconnect* computes the implicit gate identifier for a channel that is mentioned in a channel-to-channel connection.

findconnect(*ch*:<channel definition>, *id*: *DefinitionAS0*): <identifier> =$_{def}$
if *id*=**env then**
let *matchingGateIds* =
{ *c.myImplicitGateIdentifier* | *c* ∈ <channel to channel connection>:
c.parentAS0 = *ch.parentAS0* ∧ *fullIdentifier0*(*c*) ∈ *c*.**s2-**<identifier>**-seq**} **in**
matchingGateIds.take
else
let *matchingGateIds* =
{ *c.myImplicitGateIdentifier* | *c* ∈ <channel to channel connection>:
c.parentAS0 = *id.refersto0* ∧ *fullIdentifier0*(*c*) ∈ *c*.**s-**<identifier>**-seq**} **in**
matchingGateIds.take
endif

The function *allSignalsIn* computes the input signals belonging to a channel-to-channel connection.

allSignalsIn(*c*: <channel to channel connection>): *DefinitionAS0** =$_{def}$
bigSeq(< *id.refersto0*.**s-**<channel path>.**s-**<signal list item>**-seq** | *id* **in** *c*.**s-**<identifier>**-seq** >) ⌒
bigSeq(< *id.refersto0*.**s-**<channel path>.**s-**<signal list item>**-seq** | *id* **in** *c*.**s2-**<identifier>**-seq** >)

The function *allSignalsOut* computes the output signals belonging to a channel-to-channel connection.

allSignalsOut(*c*: <channel to channel connection>): *DefinitionAS0** =$_{def}$
bigSeq(< *id.refersto0*.**s-**<channel path>.**s-**<signal list item>**-seq** | *id* **in** *c*.**s-**<identifier>**-seq** >) ⌒
bigSeq(< *id.refersto0*.**s-**<channel path>.**s-**<signal list item>**-seq** | *id* **in** *c*.**s2-**<identifier>**-seq** >)

Mapping

| < <channel to channel connection>(*,*) > => *empty*

F2.2.7.3 Signal

Abstract syntax

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

Signal-parameter :: *Aggregation-kind Sort-reference-identifier*

Concrete syntax

<signal definition list> :: <signal definition>+

<signal definition> :: <type preamble>
<signal<name>
[<formal context parameters>]
[<virtuality constraint>]
[<specialization>]
[<sort list> | <named fields sort list> | <sort<identifier>]

<named fields sort list> :: [<visibility>]
{ [<visibility> <aggregation kind> <field<name> <sort> }+

<sort list> :: { <aggregation kind> <sort> }+

<as signal> :: <signal<identifier>

Further study is needed for the handling of <as signal>.

Conditions on concrete syntax

∀*sdi*∈<signal definition>:*sdi.specialization0*.**s-**<type expression>.**s-**<base type>.*idKind0* = **signal**

The <base type> part of <specialization> must be a <signal identifier>.

∀*sid*∈<identifier>:
sid.parentAS0∉<type expression>∧ *sid.parentAS0*∉<signal constraint>⇒
¬*getEntityDefinition0*(*sid*, **signal**).*isAbstractType0*

An abstract signal can only be used in specialization and signal constraints.

Transformations

< <signal definition list>(*pre*, < *item* > ⌒ *rest*) > **provided** *rest* ≠ *undefined*
=1=> < <signal definition list>(*pre*, < *item* >), <signal definition list>(*pre*, *rest*) >

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

Mapping to abstract syntax

| <signal definition list>(*, < *item* >) => *Mapping*(*item*)
| <signal definition>(*name*,*,*,*,*sortlist*)
=> **mk-***Signal-definition*(*Mapping*(*name*),
if *sortlist*= *undefined* **then** *empty* **else** *Mapping*(*sortlist*) **endif**)

F2.2.7.4 Signal list definition

Concrete syntax

<signal list definition> ::
<interface<name> <signal list>

<signal list> =
<signal list item>+

<signal list item> ::
<signal<identifier>
| <timer<identifier>
| <interface<identifier>
| <remote procedure<identifier>
| <remote variable<identifier>

Conditions on concrete syntax

∀*siglistDef*∈<signal list definition>: ¬*isSiglistContaining0*(*siglistDef*, *siglistDef*)

The <signal list definition> must not contain the <signal list identifier> defined by the <signal list definition> either directly or indirectly (via another <signal list identifier>).

Transformations

<signal list definition>(*n*,*sl*)
=9=>
<interface definition>(*empty*, *undefined*,
<interface heading>(*n*, *empty*, *undefined*), *undefined*, <entity in interface>(<interface use list>(*sl*)))

A <signal list definition> is an alternative concrete syntax, and is transformed into an <interface definition> using the keyword **interface** with no <package use clause>, no <virtuality>, an <interface heading> containing the <interface name> (but no <formal context parameters> or <virtuality constraint>), no <interface type expression> specialization and no <entity in interface>. The <interface definition> has an <interface use list> with a <signal list> that is the same as the <signal list> of the <signal list definition>.

< <signal list item>(*id*) > **provided** *id.refersto0* ∈ <signal list definition>
=8=> *id.refersto0*.**s-**<signal list item>**-seq**

Every <signal list identifier> is replaced by the list of signals of its definition.

Mapping to abstract syntax

| < <signal list item>(*id*) > => *Mapping*(*id*)

Auxiliary functions

The function *isSiglistContaining0* is used to determine if a signal list contains another signal list, either directly or indirectly.

isSiglistContaining0(*sld1*: <signal list definition>, *sld2*: <signal list definition>): *BOOLEAN* =$_{\text{def}}$
∃*sid*∈<identifier>: *sid.parentAS0.parentAS0= sld1*∧*sid.idKind0* = **signallist**∧
(*getEntityDefinition0*(*sid,* **signallist**) = *sld2*∨
(∃ *sld3*∈<signal list definition>:
isSiglistContaining0(*sld1*, *sld3*)∧ *isSiglistContaining0*(*sld3*, *sld2*)))

F2.2.7.5 Remote procedures

Concrete syntax

<remote procedure definition> ::
<remote procedure><name> <procedure signature>

<remote procedure call> :: <remote procedure call body>

<remote procedure call body> ::
<remote procedure><identifier> <actual parameters> <communication constraints>

<timer communication constraint> ::
<timer<identifier> [<variable>] [<variable>]* [<connector<name>]

Conditions on concrete syntax

∀*pd*∈<procedure definition>:
let *rpi* = *pd*.**s-**<procedure heading>. **s-**<procedure preamble>.**s-**<exported>.**s-**<identifier> **in**
let *rpd* =*getEntityDefinition0*(*rpi*, **remote procedure**) **in**
pd.isExported0 ∧ *rpi*≠ *undefined*⇒
isSameProcedureAndSignature0(*pd, rpd*.**s-**<procedure signature>)
endlet

The <remote procedure identifier> following **as** in an exported procedure definition must denote a <remote procedure definition> with the same signature as the exported procedure.

∀*pd*∈<procedure definition>:
let *rpi* = *pd*.**s-**<procedure heading>. **s-**<procedure preamble>.**s-**<exported>.**s-**<identifier> **in**
let *rpd* =*getEntityDefinition0*(*pd.name0,* **remote procedure**) **in**
pd.isExported0 ∧ *rpi* = *undefined*⇒
(*rpd* ≠ *undefined*∧*isSameProcedureAndSignature0*(*pd, rpd*.**s-**<procedure signature>))
endlet

In an exported procedure definition with no **as** clause, the name of the exported procedure is implied and the <remote procedure definition> in the nearest surrounding scope with same name is implied.

∀*rpc*∈<remote procedure call>:
(**let** *d* = *parentAS0ofKind*(*rpc,* <agent definition> ∪<agent type definition>) **in**
rpc.**s-**<identifier>∈*d.validOutputSignalSet0*
endlet)

A remote procedure mentioned in a <remote procedure call> must be in the complete output set of an enclosing agent type or agent set.

∀*exp*∈<expression>: ∀*callBody*∈<remote procedure call body>:
exp. *parentAS0.parentAS0*∈*callBody* ∧ *exp.staticSort0* ≠ "Pid" ⇒

(**let** *def* = *getEntityDefinition0*(*exp.staticSort0*, **sort**) **in**
let *pd* = *getEntityDefinition0*(*callBody*.**s-**<identifier>, **remote procedure**) **in**
def ∈ <interface definition> ∧ *isDefinedIn0*(*pd*, *def*)
endlet)

If <destination> in a <remote procedure call body> is a <pid expression> with a sort other than Pid, then the <remote procedure identifier> must represent a remote procedure contained in the interface that defined the pid sort.

∀(*cc*∈<communication constraints>: (|{*d*∈<destination>: *d* **in** *cc*}| > 1)) ⇒
cc.parentAS0∉<remote procedure call body>∪<import expression>)

A <communication constraints> in a <remote procedure call body> or <import expression> shall contain no more than one <destination>.

∀*tc*∈<timer communication constraint>: *tc.parentAS0.parentAS0*∉<output body>

The <communication constraints> in an <output body> shall not contain a <timer communication constraint>.

∀(*tc*∈<timer communication constraint>: (|{*tc* **in** *tc.parentAS0*}| ≤ 1)) ⇒
tc.parentAS0.parentAS0∉ (<remote procedure call body>∪<import expression>):

The <communication constraints> in a <remote procedure call body> or <import expression> shall contain no more than one <timer communication constraint>.

Transformations

A remote procedure call

– **call** Proc(apar) **to** destination **timer** timerinfo **via** viapath

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

let *nn* = *newName* **in**
< *r*=<remote procedure definition>(*, *sign*) >
provided *r.implicitName* = *undefined*
=16=> (
< *r*, <signal definition list>(*undefined*,
< <signal definition>(*nn*⌒"CALL",*empty*,*undefined*, *undefined*,<"Integer">) >),
<signal definition list>(*undefined*,
< <signal definition>(*nn*⌒"REPLY",*empty*,*undefined*,*undefined*,<"Integer">) >) >
and
r.implicitName:= *nn*
endlet

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

< <channel definition>(*n*, *delay*,

```
    <channel path>(ep1, ep2,
            sigs1 ⌒ <signal list item>(remote procedure, i=<identifier>(q, n)) ⌒ sigs2),
        path2, n2) >
provided i.refersto0.implicitName ≠ undefined
=17=>
    <   <channel definition>
            (   n, delay,
                <channel path>
                    (   ep1, ep2,
                        sigs1 ⌒ < <identifier>(q, i.refersto0.implicitName ⌒ "CALL") > ⌒ sigs2
                    ),
                path2, n2
            ),
        <channel definition>
            (   newName, delay,
                <channel path>(ep2, ep1, < <identifier>(q, i.refersto0.implicitName ⌒ "REPLY") > )
                undefined, undefined
            ),
    >
```

On each channel mentioning the remote procedure, the remote procedure is replaced by pCALL. For each such channel, a new channel is added in the opposite direction; this channel carries the signal pREPLY. The new channel has the same delaying property as the original one.

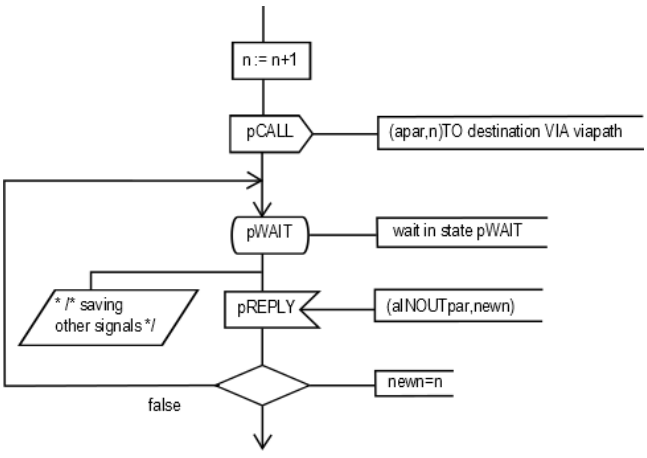
```
let nn=newName in
let varN = nn ⌒ "N" in
let varNewN = nn ⌒ "NewN" in
r=<remote procedure call>(<remote procedure call body>(id, params, constr))
=17=> <procedure call>(<procedure call body>(undefined, <identifier>(undefined, nn), empty))
and
    let varDefs = <
        <variable definition>(undefined, <
            <variables of sort>(< <variables of sort gen name>(varN, undefined) >, "Integer", "1")
        >),
        <variable definition>(undefined, <
            <variables of sort>
                (< <variables of sort gen name>(varNewN, undefined) >, "Integer", undefined)
        >) >
    in
    let timerInput = <
        <input part>(undefined,
            < <stimulus>( tid, params)   >,
            undefined,
            <terminator>(undefined, <join>(tconnect in constr:tconnect ∈ <connector name>)) >)
        | tid in constr: tid ∈ <identifier> >
    in
    let procDef = <internal procedure definition>(empty,
            <procedure heading>(<procedure preamble>(undefined, undefined), undefined, nn,
                undefined, undefined, undefined, undefined, undefined, undefined),
            empty,
            <procedure body>(undefined,
                <start>(undefined, undefined, undefined,
                    <transition action items>(<
                        <action>(undefined,
                            <task>(<assignment>(varN, <operator application>("+", < varN, "1" >)))),
                        <action>(undefined,
                            <output>(<output body>(
                                <output body item>(id.refersto0.implicitName⌒"CALL",
                                        params⌒varN),
```

```
                    constr)),
                undefined),
            >),
            <terminator>(undefined,
                <nextstate>(<nextstate body gen name>
                    (id.refersto0.implicitName⌒"WAIT", undefined, undefined))
            )
        )), <
            <state>(< id.refersto0.implicitName⌒"WAIT" >, undefined, <
                <save part>(undefined, <asterisk>),
                <input part>(undefined,
                    < <stimulus>( id.refersto0.implicitName⌒"REPLY", inoutpars ⌒ varNewN)
                    >,
                    undefined,
                    <transition>(
                        <decision>(<operator application>("=", < varNewN, varN >),
                          <answer>("true", empty)
                          <answer>("false", <terminator>(
                                <nextstate>( id.refersto0.implicitName⌒""WAIT")))
                        ),
                        <terminator>(<return>(undefined))
                    ) ) >) ⌒ timerInput) >)
in
items=getEntities(r)
=> varDefs ⌒ < procDef > ⌒ items
and
< currState = parentAS0ofKind(r, <state>)
=> <  currState,
      <state>  (  <asterisk state list>(empty),
               <  <input part>(undefined,<id.refersto0.implicitName⌒"REPLY">, undefined,
                    <terminator>(undefined, <dash nextstate>()))
               >
               )
  >
```

a) For each imported procedure, two implicit anonymous Integer variables, n and newn, are defined, and n is initialized to 0.

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

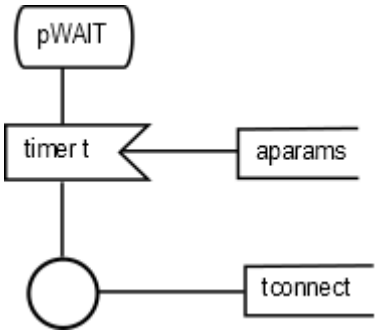
– The <remote procedure call> is transformed as below.



– where apar is the list of actual parameters except actual parameters corresponding to out parameters, and aINOUTpar is the modified list of actual in/out-parameters and

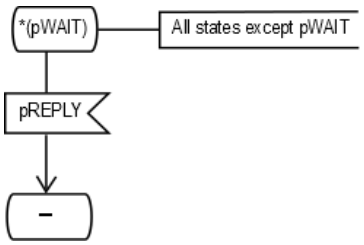
out-parameters, including an additional parameter if a value returning remote procedure call is transformed.

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



  – where t is the <timer identifier> in the <timer communication constraint>; aparams is the optional list of optional <variable> items given after the <timer identifier> in the <timer communication constraint>;
  – tconnect is the <connector name> if one is given in the <timer communication constraint>; otherwise tconnect is the name of the timer.

– In all states of the agent except pWAIT



– is inserted.

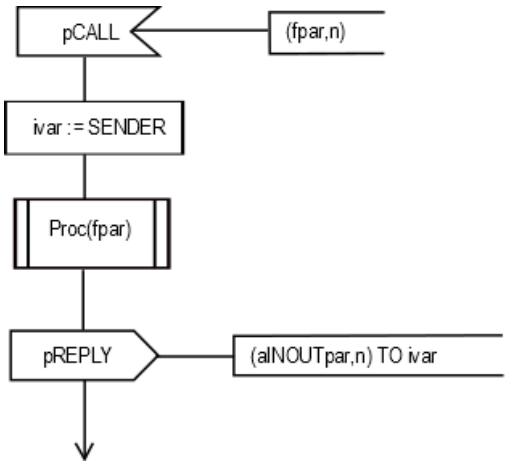
```
let n=newName in
let ivar = newName in
let res = newName in
i=<input part>(< <stimulus>(rpc) >, trans) provided rpc.refersto0 ∈ <remote procedure definition>
=17=>
    let varDefs = <
        <variable definition>(undefined, <
            <variables of sort>(< <variables of sort gen name>(n, undefined) >, "Integer", undefined)
        >),
        <variable definition>(undefined, <
            <variables of sort>
                (< <variables of sort gen name>(ivar, undefined) >, "Pid", undefined)
        >) > ⌒
        if rpc.refersto0.s-<procedure signature>.s-<result> = undefined then
            empty
        else
            <variable definition>(undefined, <
                <variables of sort>(< <variables of sort gen name>(res, undefined) >,
                    rpc.refersto0.s-<procedure signature>.s-<result>, undefined)
        >)
        endif ⌒
        remoteProcParamsDef(rpc)
    in
    items=getEntities(i)
```

```
    => varDefs ⌒ items
    endlet // varDefs
    and
    let fpar = remoteProcParams(rpc) in
        <input part>(< rpc.refersto0.implicitName ⌒ "CALL", fpar ⌒ n >,
            <transition>( <
                <task>(<assignment>     (ivar,
                            <operand5>(undefined, <sender expression>())
                )),
                if rpc.refersto0.s-<procedure signature>.s-<result> = undefined then
                    <procedure call>(<procedure call body>(undefined, rpc, fpar))
                else
                    <task>(<assignment>(res,
                                <value returning procedure call>(
                                  <procedure call body>(undefined, rpc, fpar)
                                )
                            ))
                endif,
                <output>(id.refersto0.implicitName⌒"REPLY",
                    aINOUTremoteProcParams(rpc)⌒varN, ivar),
                trans.s-<action>-seq, trans.s-<terminator>
                > )
    )
    and
    states=i.parentAS0.parentAS0.s-<state>-seq
    =>
        <   if handled (rpc, s.s-<state list>.head.name0, states) then s
            else
                <state>(s.s-<state list>, s.s-implicit ⌒
                        <input part>(rpc.refersto0.implicitName ⌒ "CALL",
                            fpar ⌒ n,
                            <transition>( <
                                <task>(<assignment>(ivar,
                                  <operand5>(undefined, <sender expression>())
                                )),
                                if rpc.refersto0.s-<procedure signature>.s-<result> ≠ undefined
                                then <procedure call>(<procedure call body>(undefined, rpc, fpar))
                                else <task>(<assignment>(res,
                                            <value returning procedure call>(
                                                <procedure call body>(undefined, rpc, fpar))
                                            )) -- task
                                endif,
                                <output>(id.refersto0.implicitName⌒"REPLY",
                                            aINOUTremoteProcParams(rpc)⌒varN, ivar),
                                <terminator>(undefined, <dash nextstate>())
                            >)
                        )
                )
            endif
            | s in states
        >
    endlet // fpar
endlet // res
endlet // ivar
endlet // n
```

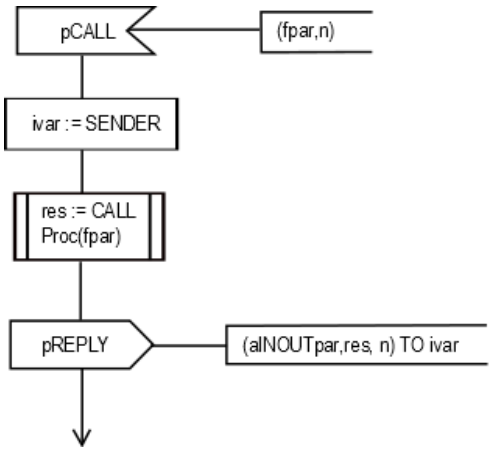
b) In the server agent, an implicit anonymous Integer variable (in this description called n) is defined for each <input area> that is a remote-procedure input. Furthermore, there is an

implicit anonymous Pid variable (in this description called ivar) for each such <input area> defined in the scope where the remote procedure input occurs. If a value returning remote procedure call is transformed, an implicit anonymous variable (in this description called res) with the same sort as <sort> in <procedure result> is defined.

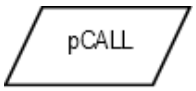
– To all <state area>s with a remote procedure input transition, the following <input area> replaces the remote procedure input and leads to the transition for the remote procedure:



– or,



– if a value returning remote procedure call was transformed.



<save part>(*virt*, < <signal list item>(**remote procedure**, *id*) >)

=17=> <save part>(*virt*, < *id.refersto0.implicitName* ⌒ "CALL" >)

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

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

Auxiliary functions

The function *implicitName* is used to store the implicitly generated name for a remote entity definition.

controlled *implicitName*: <remote procedure definition> ∪ <remote variable definition> → <name>

The function *getEntities* is used to get the entity definitions of the enclosing scope unit.

getEntities(*n*: *DefinitionAS0*): *ENTITYDEFINITION0** =def
if *n* = *undefined* **then** *undefined* **else**
case *n* **of**
| <agent type definition> => *n*.**s-**<agent structure>.**s-**<agent structure>.**s-**<entity in agent>**-seq**
| <agent definition> => *n*.**s-**<agent structure>.**s-**<agent structure>.**s-**<entity in agent>**-seq**
| <package definition>(*, *, *entities*) => *entities*
| <internal procedure definition>(*, *, *entities*, *) => *entities*
otherwise *n.parentAS0.getEntities*
endcase
endif

The function *handled* checks whether a remote procedure call is explicitly handled within a state.

handled(*rpc*: <identifier>, *state*: <name>, *states*: <state>*): *BOOLEAN* =def
∃ *s* ∈ *states*: *state* ∈ { *si.name0* | *si* ∈ *s*.**s-**<state list>.*toSet* } ∧
(*rpc* ∈ **⋃**{ { *si*.**s-**<signal list item>.**s-**<identifier> | *si* ∈ *input*.**s-**<input list>.*toSet* }
| *input* ∈ *s*.**s-implicit**: *s* ∈ <input part> } ∨
rpc ∈ **⋃**{ { *si*.**s-**<identifier> | *si* ∈ *save*.**s-**<save item>.*toSet* }
| *save* ∈ *s*.**s-implicit**: *s* ∈ <save part> })

Determine if an <internal procedure definition> and a <procedure signature> are matching.

isSameProcedureAndSignature0(*pd*:<internal procedure definition>, *ps*: <procedure signature>):*BOOLEAN* =def
let *fpl* = *ps.procedureSignatureParameterList0* **in**
let *fpl'* = *pd.procedureFormalParameterList0* **in**
(*fpl.length* = *fpl'.length*) ∧
(∀*i*∈1..*fpl.length*:
(*fpl*[*i*].**s-**<parameter kind> = *fpl'*[*i*].*parentAS0.parentAS0*.**s-**<parameter kind>) ∧
isSameSort0(*fpl*[*i*].**s-**<sort>, *fpl'*[*i*].*parentAS0*.**s-**<sort>))∧
isSameResult0(*pd*.**s-**<procedure heading>.**s-**<procedure result>, *ps*.**s-**<result>))
endlet

Determine if two results are matching.

isSameResult0(*r*: <result>∪<procedure result>∪<operation result>,
r': <result>∪<procedure result>∪<operation result>): *BOOLEAN* =def
isSameSort0(*r*.**s-**<sort>,*r'*.**s-**<sort>)

The function *remoteProcParamsDef* produces a list of variable definitions for the variables to hold the values passed via the CALL and REPLY signals for a remote procedure call.

remoteProcParamsDef(*rpc*:<signal list item>):<variable definition>*=def
// The body of this functon requires further study.

The function *remoteProcParams* produces a list of variables to hold the values passed via the CALL signal for a remote procedure call before calling the remote procedure.

remoteProcParams(*rpc*:<signal list item>):<variable>*=def
// The body of this functon requires further study.

The function *aINOUTremoteProcParams* produces variable list for the values passed via the REPLY signal of a remote procedure call with the list being in the order of each IN/OUT parameter, followed by the result parameter if there is one.

aINOUTremoteProcParams(*rpc*:<signal list item>):<variable>*=def
// The body of this functon requires further study.

F2.2.7.6 Remote variable

Concrete syntax

<remote variable definition> :: <remote variables of sort>+

<remote variables of sort> ::
<remote variable><name>+ <sort> [**nodelay**]

<export statement> :: <export body>

<export body> :: <variable><identifier>+

Conditions on concrete syntax

∀*v* ∈ <variables of sort gen name>:
let *rvd* = *getEntityDefinition0*(*v*.**s-**<identifier>, **remote variable**) **in**
v.isExported0∧ *v*.**s-**<identifier> ≠*undefined*⇒*isSameSort0*(*v.parentAS0*.**s-**<sort>, *rvd*.**s-**<sort>)
endlet

The <remote variable identifier> following **as** in an exported variable definition must denote a <remote variable definition> of the same sort as the exported variable definition.

∀*v* ∈ <variables of sort gen name>:
let *rvd* =*getEntityDefinition0*(*v*.**s-**<name>, **remote variable**) **in**
v.isExported0∧*v*.**s-**<identifier> = *undefined*⇒
(*rvd* = *undefined* ∧ *isSameSort0*(*v.parentAS0*.**s-**<sort>, *rvd*.**s-**<sort>))
endlet

In the case where there is no **as** clause, the remote variable definition in the nearest enclosing scope unit with the same name and sort as the exported variable definition is denoted.

∀*exp*∈<import expression>: *exp*.**s-**<identifier>∈
(**let** *d*=*parentAS0ofKind*(*exp*, <agent definition> ∪<agent type definition>) **in**
exp.**s-**<identifier>∈*d.validOutputSignalSet0*
endlet)

A remote variable mentioned in an <import expression> must be in the complete output set of an enclosing agent type or agent set.

∀*vid* ∈ <identifier>: *vid.parentAS0*∈<export body>⇒*getEntityDefinition0*(*vid*,**variable**).*isExported0*

The <variable identifier> in <export statement> must denote a variable defined with **exported**.

∀*exp*∈<expression>: ∀*importExp*∈<import expression>:
exp. *parentAS0* = *importExp* ∧ *exp.staticSort0* ≠ "Pid" ⇒
(**let** *def* = *getEntityDefinition0*(*exp.staticSort0*, **sort**) **in**
let *pd* = *getEntityDefinition0*(*importExp*.**s-**<identifier>, **remote variable**) **in**
def ∈ <interface definition> ∧ *isDefinedIn0*(*pd*, *def*)
endlet)

If <destination> in an <import expression> is a <pid expression> with a sort other than Pid, then the <remote variable identifier> must represent a remote variable contained in the interface that defined the pid sort.

Transformations

An import operation is modelled by exchange of implicitly defined signals. When a remote variable is conveyed on explicit channels, the **nodelay** keyword from the <remote variable definition> is ignored. The importer sends a signal to the exporter, and waits for the reply. In response to this signal the exporter sends a signal back to the importer with the result contained in the implicit copy of the exported variable.

let *nn* = *newName* **in**
< *v*= <variable definition>(**exported**, < <variables of sort>(< *n* >, *sort*, *const*) >) >
provided *v.implicitName* = *undefined*

=16=> < *v*, <variable definition>(*undefined*, < <variables of sort>(< *nn* >, *sort*, *const*) >) >
and
v.*implicitName*:= *nn*

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

let *nn* = *newName* **in**
< *r*=<remote variable definition>(< <remote variables of sort>(< *n* >, *sort*, *delay*) >) >
provided *r*.*implicitName* = *undefined*
=16=> < *r*,
<signal definition list>(*undefined*,
< <signal definition>
(*n* ⌒ "QUERY", *empty*, *undefined*, *undefined*, < "Integer" >) >),
<signal definition list>(*undefined*,
< <signal definition>
(*n* ⌒ "REPLY", *empty*, *undefined*, *undefined*, < *sort*, "Integer" >) >) >
and
r.*implicitName*:= *nn*

There are two implicit <signal definition list>s for each <remote variable definition> in a system definition. The <signal name>s in these <signal definition>s are denoted by xQUERY and xREPLY respectively, where x denotes the <name> of the <remote variable definition>. The signals are defined in the same scope unit as the <remote variable definition>. The signal xQUERY has an argument of the predefined sort Integer and xREPLY has arguments of the sort of the variable and Integer. The implicit copy of the exported variable is denoted by imcx.

< <channel definition>(*n*, *delay*,
<channel path>(*ep1*, *ep2*,
sigs1 ⌒ <signal list item>(**remote variable**, *i*=<identifier>(*q*, *n*)) ⌒ *sigs2*),
path2, *n2*) >
provided *i*.*refersto0*.*implicitName* ≠ *undefined*
=16=>
< <channel definition>(*n*, *delay*,
<channel path>(*ep1*, *ep2*, *sigs1* ⌒ < <identifier>(*q*, *i*.**s-**<name> ⌒ "QUERY") > ⌒ *sigs2*),
path2, *n2*),
<channel definition>(*newName*, *delay*,
<channel path>(*ep2*, *ep1*, < <identifier>(*q*, *i*.**s-**<name> ⌒ "REPLY") >),
undefined, *undefined*) >

On each channel mentioning the remote variable, the remote variable is replaced by xQUERY. For each such channel, a new channel is added in the opposite direction; this channel carries the signal xREPLY. In the case of a channel, the new channel has the same delaying property as the original one.

let *nn*=*newName* **in**
let *varN* = *nn* ⌒ "N" **in**
let *varNewN* = *nn* ⌒ "NewN" **in**
r=<import expression>(*id*, *constr*))
=17=> <value returning procedure call>(
<procedure call body>(*undefined*, <identifier>(*undefined*, *nn*), *empty*))
and
let *varDefs* =
< <variable definition>(*undefined*, <
<variables of sort>(< <variables of sort gen name>(*varN*, *undefined*) >, "Integer", "1") >),
<variable definition>(*undefined*, <
<variables of sort>(< <variables of sort gen name>(*varNewN*, *undefined*) >,
"Integer", *undefined*)

```
>) >
in
let timerInput = <
    <input part>(undefined,
        < <stimulus>( tid, params)  >,
        undefined,
        <terminator>(undefined, <join>(tconnect in constr:tconnect ∈ <connector name>)) >)
    | tid in constr: tid ∈ <identifier> >
in
let procDef = <internal procedure definition>(empty,
        <procedure heading>(<procedure preamble>(undefined, undefined), undefined, nn,
            undefined, undefined, undefined, undefined,
            <procedure result>(undefined, id.refersto0.s-<sort>), undefined),
        empty,
        <procedure body>(undefined,
            <start>(undefined, undefined, undefined,
                <transition action items>(<
                    <action>(undefined,
                        <task>(<assignment>(varN, <operator application>("+", < varN, "1" >)))),
                    <action>(undefined,
                        <output>(<output body>(
                            <output body item>(id.refersto0.implicitName ⌒ "QUERY",
                              params ⌒ varN), constr)),
                        undefined),
                >),
                <terminator>(undefined,
                    <nextstate>(<nextstate body gen name>("WAIT", undefined, undefined))
                )
        )), <
            <state>(< "WAIT" >, undefined, <
                    <save part>(undefined, <asterisk>),
                    <input part>(undefined,
                        < <stimulus>( id.refersto0.implicitName⌒"REPLY", < id > ⌒ varNewN) >,
                        undefined,
                        <transition>(
                            <decision>(<operator application>("=", < varNewN, varN >),
                              <answer>("true", empty)
                              <answer>("false", <terminator>(<nextstate>("WAIT")))
                            ),
                            <terminator>(<return>(undefined))
                        ) ) >),
                    timerInput) >)
in
items=getEntities(r)
=> varDefs ⌒ < procDef > ⌒ items
and
< currState = parentAS0ofKind(r, <state>)
=> < currState,
        <state>(<asterisk state list>(empty),
            < <input part>(undefined, < "REPLY" >, undefined,
                <terminator>(undefined, <dash nextstate>()))>
```

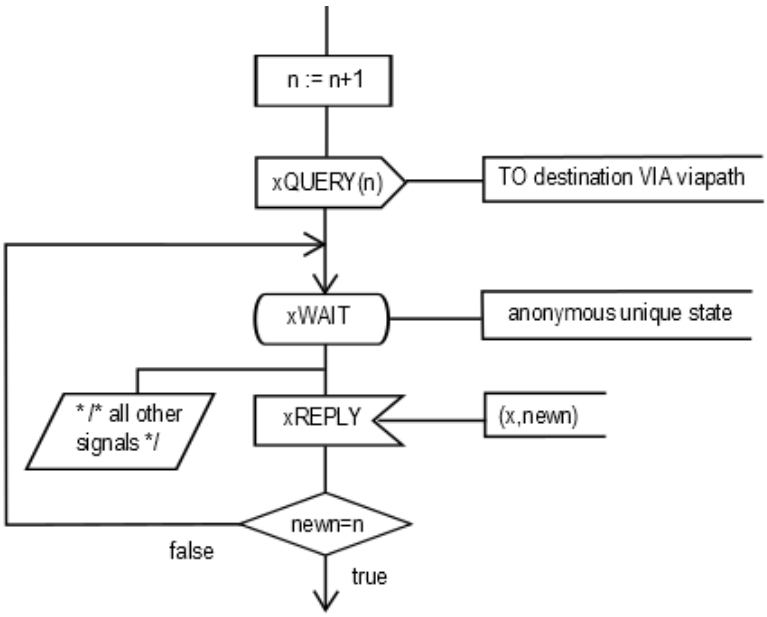
a) Importer

– For each imported variable, two implicit Integer variables n and newn are defined, and n is initialized to 0. In addition, an implicit variable x of the sort of the remote variable is defined.

– The <import expression>

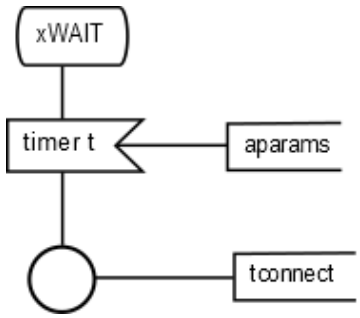
import (x, destination via via-path)

is transformed to the following, where the to clause is omitted if the destination is not present, and the via clause is omitted if it is not present in the original expression:



– In all other states, xREPLY is saved.

– Additionally, the following will be inserted for every timer t that is included in <communication constraints>:

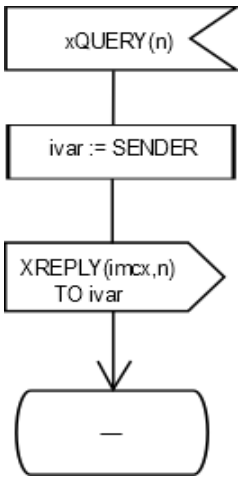


where t is the <timer identifier> in the <timer communication constraint>; aparams is the optional list of optional <variable> items given after the <timer identifier> in the <timer communication constraint>; tconnect is the <connector name> if one is given in the <timer communication constraint>; otherwise tconnect is the name of the timer.

i=<input part>(*rv*, *trans*) **provided** *rpc.refersto0* ∈ <remote variable definition>
=17=> <input part>(*rv*.s-<name> ⌒ "QUERY", *n*, <transition>(<
<task>(<assignment>(*ivar*, <operand5>(*undefined*, <sender expression>()))),
<output>(<output body>(<
<output body item>(*rv*.s-<name> ⌒ "QUERY", *rpc.refersto0.implicitName*) >,
< <destination>(*ivar*)>),
rv.s-<name> ⌒ "QUERY")
>
⌒ *trans*.**s-**<action>**-seq**, *trans*.**s-**<terminator>))
))

b) Exporter

– To all <state area>s of the exporter, excluding implicit states derived from import, the following <input area> is added:



For each such state, an implicit anonymous variable of sort Pid (in this description called ivar) and an implicit anonymous variable of type Integer (in this description called n) are defined.

<export statement>(<export body>(< *id* >))
=17=>
<task>(<assignment>(*id.refersto0.implicitName*, *id*))

The <export statement>

export x

is transformed to the following:

task imcx:= x;

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

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

F2.2.7.7 Communication path encoding rules, encode and decode

Abstract syntax

Encoding-rules	::	*Rules-identifier*
Encoding-expression	::	*Signal-identifier* [*Expression*]* *Encoding-path*
Encoding-path	::	{ *Gate-identifier* | *Data-type-identifier Rules-identifier* }
Decoding-expression	::	*Expression* *Encoding-path*
Rules-identifier	=	*Literal-identifier*

Concrete syntax

<encoding rules> :: <rules identifier>

<rules identifier> :: <literal>

<encoding expression> :: { <signal<identifier> [<actual parameters>] | <expression> }[<encoding path>]

<encoding path> :: { <channel<identifier> | <gate<identifier> | <interface<identifier> <rules<identifier> }

<decoding expression> :: <expression> [<encoding path>]

Conditions on concrete syntax

Further study required.

Mapping to abstract syntax

Further study required.

F2.2.8 Behaviour

F2.2.8.1 Start

Abstract syntax

State-start-node	::	*Transition*
Procedure-start-node	::	*Transition*

Concrete syntax

<start> :: [<virtuality>] [<state entry point<name>] <transition>

Conditions on concrete syntax

∀*s*∈<start>:
(*s*.**s**-<name> ≠ *undefined*) ⇒
(*s.surroundingScopeUnit0*∈<composite state>)∨
(*s.surroundingScopeUnit0*∈<composite state type definition>)

If <state entry point name> is given in a <start>, the <start> must be the <start> of a <composite state>.

Mapping to abstract syntax

| *s*=<start>(*,*entry*,*trans*)
=> **if** *s.parentAS0* ∈ <procedure body> **then mk-***Procedure-start-node*(*Mapping*(*trans*))
else mk-*State-start-node*(*Mapping*(*entry*), *Mapping*(*trans*)) **endif**

F2.2.8.2 State

Abstract syntax

State-node	::	*State-name* *Save-signalset* *Input-node*-**set** {*Spontaneous-transition*-**set** *Continuous-signal*-**set** | *Composite-state-type-identifier Connect-node*-**set** } [*State-timer*]
State-timer	::	*Time-expression* *Timer-identifier* *Expression** *Transition*

Conditions on abstract syntax

∀*sn, sn'* ∈ *State-node*:(*sn*≠*sn'*)∧(*sn*.*parentAS1*=*sn'*. *parentAS1*)⇒(*sn*.**s**-*State-name*≠*sn'*.**s**-*State-name*)

*State-node*s within a *State-transition-graph* or *Procedure-graph* must have different *State-name*.

∀ *sn* ∈ *State-node*: ∀*in*,*in'*∈*Input-node*:
(*in*≠*in'*)∧(*in*. *parentAS1*=*sn*)∧(*in'*. *parentAS1*=*sn*)⇒*in*.**s**-*Signal-identifier*≠*in'*.**s**-*Signal-identifier*

The *Signal-identifier*s in the *Input-node*-**set** must be distinct.

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

Concrete syntax

<state> ::
    <state list>
        { <input part>
        | <priority input>
        | <save part>
        | <spontaneous transition>
        | <continuous signal>
        | <connect part> }*
        [<state timer part>]

<state list> =
    {<basic state name> | <typebased composite state> | <composite state list item>}+
  | <asterisk state list>

<basic state name> = <basic state<name>

<composite state list item> :: <composite state name> <nextstate parameters>

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

<asterisk state list> :: <state<name>*

<state timer part> ::
    [<virtuality>] <state timer> <transition>

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

NOTE – Further study is needed to model state timers.

Conditions on concrete syntax

$\forall bs \in$ <state>:$\forall sn \in$ <name>:
  (*sn.parentAS0* = *bs*.**s**-<state list>$\wedge$ *bs*.**s**-<state list>$\in$**s**-<asterisk state list>) $\Rightarrow$
    ($\forall sn' \in$ <name> (*sn.parentAS0* = *sn'.parentAS0*) $\Rightarrow$ (*sn* $\neq$ *sn'*)) $\wedge$
      (*sn*$\in$*bs.surroundingScopeUnit0.stateNameSet0*)

The <state name>s in an <asterisk state list> must be distinct and must be contained in other <state list>s in the enclosing body or in the body of a supertype.

$\forall r,r' \in$ <composite state reference>:
  (*r.referencedDefinition0* = *r'.referencedDefinition0*) $\Rightarrow$
    (*r.parentAS0* = *r'.parentAS0*)$\wedge$
    ($\exists!csa \in$ <state>:$\exists sn \in$ <name>:
      (*sn*$\in$*csa.stateNameSet0*)$\wedge$ //*sn* is a state name of *csa*.
      (*sn* = *r.surroundingScopeUnit0.entityName0*))// surrounding scope of r is a composite state

A <composite state reference> to the same composite state must only occur in one of the <composite state application>s in the surrounding state machine.

Transformations

< <state>(< *s* > $\frown$ *rest*, *triggers*) > **provided** *rest* $\neq$ *empty*
=1=> < <state>(< *s* >, *triggers*), <state>(*rest*, *triggers*) >

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

< <state>(< *s* >, *exc1*, *triggers1*) > $\frown$ *rest* $\frown$ < <state>(< *s* >, *exc2*, *triggers2*) >

=13=> *rest*$\frown$< <state>(< *s* >, **if** *exc1* $\neq$ *undefined* **then** *exc1* **else** *exc2* **endif**, *triggers1* $\frown$ *triggers2*) >

When several <state> items contain the same <state name>, these <state> items are concatenated into one <state> having that <state name>.

b=<state>(<asterisk state list>(*exceptStates*), *triggers*)
=13=>
<state>(< *s* **in** *b.surroundingScopeUnit0.stateNameSet0*: *s* ∉ *exceptStates.toSet* >, *triggers*)

A <state> with an <asterisk state list> is transformed to a list of <state>s, one for each <state name> of the body in question, except for those <state name>s contained in the <asterisk state list>.

Mapping to abstract syntax

| <state>(< <basic state name>(*name*,) >, *triggers*)
=> **mk-***State-node*(*Mapping*(*name*),
{ *t* ∈ *Mapping*(*triggers*).*toSet*: *t* ∈ *Save-signalset*},
{ *t* ∈ *Mapping*(*triggers*).*toSet*: *t* ∈ *Input-node*},
{ *t* ∈ *Mapping*(*triggers*).*toSet*: *t* ∈ *Spontaneous-transition*},
{ *t* ∈ *Mapping*(*triggers*).*toSet*: *t* ∈ *Continuous-signal*}, *undefined*)

| <state>(< <composite state list item>(*name*, *undefined*) >, *triggers*)
=> **mk-***State-node*(*Mapping*(*name*),
{ *t* ∈ *Mapping*(*triggers*).*toSet*: *t* ∈ *Save-signalset*},
{ *t* ∈ *Mapping*(*triggers*).*toSet*: *t* ∈ *Input-node*},
{ *t* ∈ *Mapping*(*triggers*).*toSet*: *t* ∈ *Spontaneous-transition*},
{ *t* ∈ *Mapping*(*triggers*).*toSet*: *t* ∈ *Continuous-signal*}, *undefined*)

| <state>(< <typebased composite state>(*name*, *parent*) >, *triggers*)
=> **mk-***State-node*(*Mapping*(*name*),
{ *t* ∈ *Mapping*(*triggers*).*toSet*: *t* ∈ *Save-signalset*},
{ *t* ∈ *Mapping*(*triggers*).*toSet*: *t* ∈ *Input-node*},
{ *t* ∈ *Mapping*(*triggers*).*toSet*: *t* ∈ *Spontaneous-transition*},
{ *t* ∈ *Mapping*(*triggers*).*toSet*: *t* ∈ *Continuous-signal*}, *Mapping*(*parent*))

Auxiliary functions

Get the set of <state name>s of a <state>, an agent definition, an agent type definition, a composite state, a composite state type definition or a procedure definition.

stateNameSet0(*d*: <state>∪<agent definition> ∪<agent type definition> ∪<composite state> ∪
<composite state type definition>∪<internal procedure definition>): <name>-**set** =$_{def}$
if (*d*∈<state>) **then**
if *d*.**s**-<state list>∈<asterisk state list>**then**
d.surroundingScopeUnit0.stateNameSet0\{*n*∈<name>: *n.parentAS0*=*d*.**s**-<state list>}
else //*d*.**s**-<state list>∈{<basic state name>|<composite state list item>|<typebased composite state>}+
{*n*∈<name>: *n.parentAS0*=*d*.**s**-<state list>}
endif
else {*n*∈<name>: ∃*s*∈<state>: (*s*∈*d.stateSet0*)∧(*n*∈*s.stateNameSet0*)}
endif

F2.2.8.3 Input

Abstract syntax

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

Conditions on abstract syntax

∀*in* ∈ *Input-node*: ∀*sd* ∈ *Signal-definition*:

sd= getEntityDefinition1(*in*.**s**-*Signal-identifier*, **signal**)⇒
(*in*.**s**-*Variable-identifier*-**seq**.*length*=*sd*.**s**-*Sort-reference-identifier*-**seq**.*length*) ∧
(∀*i* ∈ 1..*in*.**s**-*Variable-identifier*.*length*:
∃*vd* ∈ *Variable-definition*: *vd=getEntityDefinition1*(*in*.**s**-*Variable-identifier*[*i*],**variable**)∧
isCompatibleTo1(*vd*.**s**-*Sort-reference-identifier*, *sd*.**s**-*Sort-reference-identifier*[*i*]))

The length of the list of optional *Variable-identifier*s must be the same as the number of items in the *Sort-reference-identifier* list in the *Signal-definition* denoted by the *Signal-identifier* and the sorts of the variables must correspond by position to the sorts of the data items that can be carried by the signal.

Concrete syntax

<input part> ::
[<virtuality>] <input list> [<enabling condition>] <transition>

<input list> =
{ <stimulus> [<in choice>] [<priority clause>]}+
| <asterisk input list> [<in choice>] [<priority clause>]
| <encoded input>

<stimulus> :: <signal list item> [<variable>]* [<via path>]

<asterisk input list> :: {}

<via path> :: <channel<identifier> |<gate<identifier>

<in choice> :: <choice<variable>

<encoded input> :: <variable> [<encoding path>]

Further study is needed for <in choice> and <priority clause>.

Further study is probably needed for <via path> in <stimulus>.

Further study is needed for <encoded input>.

Conditions on concrete syntax

∀*s*∈<state>: |*s.asteriskInputListSet0*| ≤ 1

A <state> may contain at most one <asterisk input list>.

∀*s*∈<state>: (*s.asteriskInputListSet0*=∅) ∨ (*s.asteriskSaveListSet0* = ∅)

A <state> must not contain both <asterisk input list> and <asterisk save list>.

∀*ip*∈<input part>: ∀*sli*∈<signal list item>:
isAncestorAS0(*ip*,*sli*)⇒
(**let** *idKind* = *sli*.**s**-<identifier>.*idKind0***in**
(*idKind*≠ **remote variable**)∧
(*idKind* = **remote procedure**∨ *idKind* = **signallist**⇒
sli.*parentAS0*.**s**-<variable>-**seq**=*empty*)
endlet)

A <signal list item> must not denote a <remote variable identifier> and if it denotes a <remote procedure identifier> or a <signal list identifier>, the <stimulus> parameters must be omitted.

Transformations

< <stimulus>(<signal list item>(**signallist**, *id*), *todo*) >
=8=> < <stimulus>(*sig*, *todo*) | *sig* **in** *id.refersto0*.**s**-<signal list item>-**seq** >

A <stimulus> whose <signal list item> is a <signal list identifier> is derived syntax for a list of <stimulus>s without parameters and is inserted in the enclosing <input list> or <priority input list>. In this list, there is a one to one correspondence between the <stimulus>s and the members of the signal list.

< <input part>(*virt*, < *stim* > ⌒ *rest*, *cond*, *trans*) > **provided** *rest* ≠ *empty*
=1=> < <input part>(*virt*, < *stim* >, *cond*, *trans*), <input part>(*virt*, *rest*, *cond*, *trans*) >

When the <stimulus>s list of an <input part> contains more than one <stimulus>, a copy of the <input part> is created for each such <stimulus>. Then the <input part> is replaced by these copies.

let *ip* = <input part>(*virt*, < <stimulus>(*item*, *vars*) >, *cond*, <transition action items>(*actions*, *term*)) **in**
let *newvars* =
    < (**if** *v* ∈ (<indexed variable> ∪ <field variable>) **then** *newName* **else** *v* **endif**) | *v* **in** *vars* > **in**
let *vardefs* = *defineNewvars0*(*vars*, *newvars*) **in**
items = *parentAS0ofKind*(*ip*, *SCOPEUNIT0*).*getEntities*
    provided { *v* ∈ *vars*.*toSet*: *v* ∈ (<indexed variable> ∪ <field variable>) } ≠ ∅
    // one or more of the stimulus variable items of a <stimulus> is an <extended variable>
=8=> *items* ⌒ *vardefs* // add implicitly defined variable definitions to the surrounding scope
and
let *newactions* = < *assignNewvarsToExtendedVars0*(*vars*, *newvars*) > ⌒ *actions* **in**
    // make new action list from constructed task before existing actions
ip
    provided { *v* ∈ *vars*.*toSet*: *v* ∈ (<indexed variable> ∪ <field variable>) } ≠ ∅
    // one or more of the stimulus variable items of a <stimulus> is an <extended variable>
=8=>
    <input part>(*virt*, < <stimulus>(*item*, *newvars*) >, *cond*, <transition action items>(*newactions*), *term*))
    // replace action list by new action list in the <input part>
endlet // newactions
endlet // vardefs
endlet // newvars
endlet // ip

When one or more of the stimulus variable items of a <stimulus> is an <extended variable> (an <indexed variable> or <field variable>), then each <extended variable> is replaced by a unique, new, anonymous implicitly declared <variable identifier> of the same sort as the original <extended variable>. Directly following the input area (<input area> or <priority input area>), a <task area> is inserted which in its <task body> contains an <assignment statement> for each of the <extended variable> items, assigning the result of the corresponding new variable to the <extended variable>. The results are assigned in the order from left to right of the <extended variable> list. This <task area> becomes the first <action area> in the <transition area> of the input area.

The following statement is handled by the dynamic semantics.

An <asterisk input list> is transformed to a list of <input part>s, one for each member of the complete valid input signal set of the enclosing <agent definition>, except for <signal identifier>s of implicit input signals introduced by the concepts in clauses 10.5 and 10.6 of [ITU-T Z.102] and for <signal identifier>s contained in the other <input list>s and <save list>s of the <state>.

Mapping to abstract syntax

| <input part>(*, < <stimulus>(*item*, *vars*) >, *cond*, *trans*)
    => **mk-***Input-node*(*undefined*, *Mapping*(*item*),
                Mapping(*vars*), *Mapping*(*cond*), *Mapping*(*trans*))

Auxiliary functions

Make an action list that is assignments of new variables to extended variables.

assignNewvarsToExtendedVars0(*varsl*:<variable>**-seq**, *newvarsl*:<variable>**-seq**):<action>**-seq**=$_{\text{def}}$
if *varsl* =*empty* **then** *empty*
else
    let *v* = *varsl*.*head* **in**
    let *newv* = *newvarsl*.*head* **in**
    if *v* ∈ (<indexed variable> ∪ <field variable>)

```
    then
    // create an assignment of the result of the corresponding newv to the extended variable v
        <action>(undefined,<task>(<assignment>(v, newv),undefined))⌢
            assignNewvarsToExtendedVars0(varsl.tail, newvarsl.tail)
    else    assignNewvarsToExtendedVars0(varsl.tail, newvarsl.tail)
    endif
    endlet // newv
    endlet // v
endif
```

Definitions for new variables that replace extended variables in a stimulus.

```
defineNewvars0(varsl:<variable>-seq, newvarsl:<variable>-seq):<variable definition>-seq=def
if varsl =empty
then empty
else
    let v = varsl.head  in
    let newv = newvarsl.head  in
    if v ∈ (<indexed variable> ∪ <field variable>)
    then
    // create a definition of the newv that corresponds to the extended variable v
        <variable definition>( <variables of sort>(PART, newv, getStaticSort0(v)))⌢
            defineNewvars0 (varsl.tail, newvarsl.tail)
    else    defineNewvars0 (varsl.tail, newvarsl.tail)
    endif
    endlet // newv
    endlet // v
endif
```

Get the <asterisk input list> for a <state>.

```
asteriskInputListSet0(s: <state>): <asterisk input list>-set =def
{ail∈<asterisk input list>: isAncestorAS0(s,ail)}
```

F2.2.8.4 Priority input

Concrete syntax

```
<priority input> ::
        [<virtuality>] <priority input list> <transition>

<priority input list> =
        <priority stimulus>+
    |   <asterisk input list> [ <priority clause> ]

<priority stimulus> = <stimulus>  [ <priority clause> ]

<priority clause> :: <stimulus>

<priority name> = <Natural<name>
```

NOTE – In [ITU-T Z.103] clause 11.4 Priority Input, *Concrete grammar* <priority name> is defined as a <integer name>, which is equivalent to AS0 <Natural<name>, because AS0 <name> includes the SDL-2010 lexical units <name>, <integer name> and <real name>.

Further study is needed for <priority stimulus>, an <asterisk input list> in a <priority input>, <priority clause> and <priority name> in a <priority input>.

Mapping to abstract syntax

```
| <priority input>(*, <priority input list>(< <stimulus>(item, vars) >), trans)
    => mk-Input-node(PRIORITY, Mapping(item), Mapping(vars), undefined, Mapping(trans))
```

F2.2.8.5 Continuous signal

Abstract syntax

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

Concrete syntax

<continuous signal> ::
[<virtuality>] <continuous expression> [<priority name>] <transition>

<continuous expression> = <Boolean<expression>

Mapping to abstract syntax

| <continuous signal>(*, *expr*, *prio*, *trans*)
=> **mk-***Continuous-signal*(*Mapping*(*expr*), *Mapping*(*prio*), *Mapping*(*trans*))

F2.2.8.6 Enabling condition

Abstract syntax

Provided-expression	=	*Boolean-expression*
Boolean-expression	=	*Expression*

Concrete syntax

<enabling condition> = <provided expression>

<provided expression>:: <Boolean<expression>

Mapping to abstract syntax

| <provided expression>(*expr*) => *Mapping*(*expr*)

F2.2.8.7 Save

Abstract syntax

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

Concrete syntax

<save part> :: [<virtuality>] <save list>

<save list> :: <save item>+ | <asterisk save list>

<asterisk save list> :: ()

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

Conditions on concrete syntax

$\forall s \in$ <state>: $|s.asteriskSaveListSet0| \leq 1$

A <state> may contain at most one <asterisk save list>.

$\forall s \in$ <state>: $(s.asteriskInputListSet0 = \varnothing) \vee (s.asteriskSaveListSet0 = \varnothing)$

A <state> must not contain both <asterisk input list> and <asterisk save list>.

Transformations

The following statement is handled by the dynamic semantics.

An <asterisk save list> is transformed to a list of <stimulus>s containing the complete valid input signal set of the enclosing <agent definition>, except for <signal identifier>s of implicit input signals introduced by the concepts in clauses 10.5 and 10.6 of [ITU-T Z.102] and for <signal identifier>s contained in the other <input list>s and <save list>s of the <state>.

Mapping to abstract syntax

| <save part>(*, < *id* >) => *Mapping*(*id*)

Auxiliary functions

Get the set of <asterisk save list> for s <state>.

asteriskSaveListSet0(*s*: <state>): <asterisk save list>-**set** =$_{def}$
{*asl*∈<asterisk save list>: *isAncestorAS0*(*s*,*asl*)}

F2.2.8.8 Implicit transition

Transformations

The following statement is handled by the dynamic semantics.

Any signal not handled by an explicit input or save is consumed by an implicit transition (a transition of an implicit <input association area>) without a change of state.

F2.2.8.9 Spontaneous transition

Abstract syntax

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

Concrete syntax

<spontaneous transition> ::
[<virtuality>] [<enabling condition>] <transition>

Mapping to abstract syntax

| <spontaneous transition>(*, *cond*, *trans*)
=> **mk**-*Spontaneous-transition*(*Mapping*(*cond*), *Mapping*(*trans*))

F2.2.8.10 Label

Abstract syntax

Free-action :: *Connector-name Transition*

Concrete syntax

<label> :: <connector name>

<connector name> = <name>

NOTE – An AS0 <connector name> is defined to be the same as an AS0 <name>, therefore the terms <connector name> and <connector<name> are interchangeable. In [ITU-T Z.101] a distinction is made between a <name> and an <integer name>, and a <connector name> can be either a <name> or an <integer name>.

<free action> :: <transition>

Conditions on concrete syntax

∀*b*∈<composite state body>∪<operation body>∪<procedure body>∪<agent body>: ∀*l*, *l'*∈<label>:
(*isAncestorAS0*(*b*,*l*) ∧ *isAncestorAS0* (*b*,*l'*) ∧ (*l* ≠ *l'*) ⇒ (*l*.**s**-<name> ≠ *l'*.**s**-<name>))

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

∀*fa*∈<free action>: ∀*l*∈<label>:
((*l* = *fa*.**s**-<transition>.**s**-<terminator>.**s**-<label>) ∨
(*l* =*fa*.**s**-<transition>.**s**-<action>-**seq**.*head*.**s**-<label>)) ⇒
(*l*.**s**-<name> = *fa*.**s**-<name>)

If the <transition string> of the <transition> in <free action> is non-empty, the first <action> must have a <label> otherwise the <terminator> must have a <label>. If present, the <connector<name> ending the <free action> must be the same as the <connector<name> in this <label>.

Auxiliary functions

The function *getLabel* extracts the first label from the transition.

getLabel(*t*: <transition>): <label> =$_{def}$
if *t*.**s**-<action> = *empty* **then** *t*.**s**-<terminator>.**s**-<label>
else *t*.**s**-<action>.*head*.**s**-<label>
endif

Transformations

< *a*, *s*=<action>(*l*, *) > ⌒ *r* **provided** *l* ≠ *undefined*
=5=> < *a* >
and
a.*parentAS0* => <transition action items>(*a*.*parentAS0*.**s**-<action>,
<terminator>(*undefined*, <join>(*l*.**s**-<name>)))
and
let *p* = *parentAS0ofKind*(*a*,<free action> ∪ <state>) **in**
< *p* > => < *p*, <free action>(< *s* > ⌒ *r*) >

If a <label> is not the first label of a <transition string>, the <transition string> is split into two parts. All <action>s preceding the <label> are preserved in the original transition, which is terminated with a <join> to the <label>. All <action>s following <label> are copied to a new <free action>, which starts with the <label>.

Mapping to abstract syntax

| <free action>(*trans*)
=> **mk**-*Free-action*(*Mapping*(*getLabel*(*trans*)), *Mapping*(*trans*))

F2.2.8.11 State machine and composite state

Abstract syntax

Composite-state-formal-parameter	=	*Agent-formal-parameter*
State-entry-point-definition	=	*Name*
State-exit-point-definition	=	*Name*
Exit-procedure-definition	=	*Procedure-definition*
Entry-procedure-definition	=	*Procedure-definition*
Named-start-node	::	*State-entry-point-name* *Transition*

Conditions on abstract syntax

(∀*d* ∈ *Entry-procedure-definition*:
(*d*.**s**-*Procedure-name*= "entry")∧(*d.formalParameterList1 = empty*)∧(*d.stateNodeSet1*=∅))

(∀*d* ∈ *Exit-procedure-definition*:
(*d*.**s**-*Procedure-name*="exit")∧(*d.formalParameterList1 = empty*)∧(*d. stateNodeSet1*=∅))

Entry-procedure-definition represents a procedure with the name entry. *Exit-procedure-definition* represents a procedure with the name exit. These procedures may not have parameters, and may only contain a single transition.

Concrete syntax

<composite state> ::
    <composite state graph> | <state aggregation>

F2.2.8.11.1 Composite state graph

Abstract syntax

Composite-state-graph :: *State-transition-graph*
[*Entry-procedure-definition*]
[*Exit-procedure-definition*]

Concrete syntax

<composite state graph> =
    <package use clause>* <composite state heading> <composite state structure>

<composite state heading> ::
    <virtuality> [<qualifier>] <composite state name> [<specialization>] <agent formal parameters>

<composite state structure> ::
    <state connection points>* <entity in composite state>*
    <composite state body>

<entity in composite state> =
    <valid input signal set>
  | <variable definition>
  | <data definition>
  | <select definition>
  | <procedure definition>
  | <procedure reference>
  | <composite state>
  | <composite state type definition>
  | <composite state type reference>
  | <gate in definition>

<composite state body> ::
    <start>* {<state> | <free action>}*

Conditions on concrete syntax

$\forall csg \in$ <composite state body>:
  $\exists! s \in$ <start>: ($s.parentAS0 = csg$) $\wedge$ (s.**s-**<name> = *undefined*)

Exactly one of the <start>s shall be unlabelled.

$\forall csd \in$ <composite state> $\cup$ <composite state type definition>:
  ($\forall pn \in$ <name>: $pn \in csd.usedEntryNameSet0 \Rightarrow pn \in csd.definedEntryNameSet0$) $\wedge$
  ($\forall pn \in$ <name>: $pn \in csd.usedExitNameSet0 \Rightarrow pn \in csd.definedExitNameSet0$)

Each additional labelled entry and exit point must be defined by a corresponding <state connection points>.

$\forall csb \in$ <composite state body>: $\forall s \in$ <state>: ($s \in csb.surroundingScopeUnit0.stateSet0$) $\wedge$
  ((s.**s-**<state list> $\notin$ <asterisk state list>)) $\Rightarrow$ $csg.surroundingScopeUnit0.startSet0 \neq \emptyset$

If a <composite state body> contains at least one <state> different from asterisk state, a <start> must be present.

$\forall cs \in$ <composite state>: $\forall vd \in$ <variable definition>:
  ($vd.surroundingScopeUnit0 = cs$) $\wedge$ ($cs.surroundingScopeUnit0 \in$ <internal procedure definition>) $\Rightarrow$
    (vd.**s-exported** = *undefined*)

<variable definition> in a <composite state>, cannot contain **exported** <variable name>s, if the <composite state> is enclosed by an <internal procedure definition>.

Transformations

<composite state body>(*empty*,
    items1 ⌒ <state>(<asterisk state list>(*undefined*), *triggers*, *undefined*) ⌒ *items2*)
provided < *i* **in** (*items1* ⌒ *items2*): (*i* ∈ <state>) > = *empty*
=8=>
  let *nn* = *newName* **in**
  let *startTrans* = <transition action items>(*empty*,
      <terminator>(*undefined*,
        <nextstate>(<nextstate body gen name>(*nn*, *undefined*, *undefined*)))) **in**
    <composite state body>(< <start>(*undefined*, *undefined*, *undefined*, *startTrans*) >,
      items1 ⌒ <state>(< *nn* >, *triggers*, *undefined*) ⌒ *items2*)
  endlet)

If the <composite state> consists of no <state>s with <state name>s but only a <state> with <asterisk>, transform the asterisk state into a <state> with an anonymous <state name> and a <start> leading to this <state>.

Mapping to abstract syntax

| <composite state>(*, <composite state heading>(*, *name*, *params*),
    <composite state structure>(*, *gates*, conns, *entities*, <composite state body>(*starts*, *items*)))
  => **mk-***Composite-state-graph*(
      mk-*State-transition-graph*(*head*(< *s* **in** *Mapping*(*starts*): (*s* ∈ *State-start-node*)>),
        { *s* ∈ *Mapping*(*items*): *s* ∈ *State-node* },
        { *s* ∈ *Mapping*(*items*): *s* ∈ *Free-action*}),
      head(< *e* **in** *Mapping*(*entities*):
          (*e* ∈ *Procedure-definition* ∧ *e*.**s-***Procedure-name* = "entry")>),
      head(< *e* **in** *Mapping*(*entities*):
          (*e* ∈ *Procedure-definition* ∧ *e*.**s-***Procedure-name* = "exit")>),
      { *s* ∈ *Mapping*(*starts*).*toSet*: *s* ∈ *Named-start-node* })

Auxiliary functions

Get the set of entry names used in a <composite state> or a <composite state type definition>.

usedEntryNameSet0(*csd*: <composite state>∪<composite state type definition>):<name>-**set**=$_{\text{def}}$
  {*n*∈<name>: *n.parentAS0*∈*csd.startSet0*}

Get the set of exit names used in a <composite state> or a <composite state type definition>.

usedExitNameSet0(*csd*: <composite state>∪<composite state type definition>):<name>-**set**=$_{\text{def}}$
  {*n*∈<name>: ∃*s*∈*csd.stateSet0*: *isAncestorAS0*(*s*, *n*)∧(*n.parentAS0.parentAS0*∈<return>)}

Get the set of entry names defined in a <composite state> or a <composite state type definition>.

definedEntryNameSet0(*csd*: <composite state>∪<composite state type definition>):<name>-**set**=$_{\text{def}}$
  {*n*∈<name>: (*n.parentAS0*∈<state entry points>)∧
    (*n.parentAS0.parentAS0*=*csd*.**s-**<composite state structure>)}

Get the set of exit names defined in a <composite state> or a <composite state type definition>.

definedExitNameSet0(*csd*: <composite state>∪<composite state type definition>):<name>-**set**=$_{\text{def}}$
  {*n*∈<name>: (*n.parentAS0*∈<state exit points>)∧
    (*n.parentAS0.parentAS0*=*csd*.**s-**<composite state structure>)}

F2.2.8.11.2 State aggregation

Abstract syntax

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

Conditions on abstract syntax

∀*pn*∈*State-entry-point-name*: (*pn.parentAS1*∈*Outer-entry-point*) ⇒
(*pn*∈ *pn.surroundingScopeUnit0.entryPointSet1*)∧
(*pn.surroundingScopeUnit0*.**s-implicit**∈*State-aggregation-node*)

The *State-entry-point-name* in the *Outer-entry-point* must denote a *State-entry-point-definition* of the *Composite-state-type-definition* where the *State-aggregation-node* occurs.

∀*pn*∈*State-entry-point-name*: (*pn.parentAS1*∈*Inner-entry-point*) ⇒
(*pn*∈*parentAS1ofKind*(*pn, State-partition*).*baseType1.entryPointSet1*)

The *State-entry-point-name* of the *Inner-entry-point* must denote a *State-entry-point-definition* of the composite state in the *State-partition*.

∀*pn*∈*State-exit-point-name*: (*pn.parentAS1*∈*Outer-exit-point*) ⇒
(*pn*∈ *pn.surroundingScopeUnit0.exitPointSet1*)∧
(*pn.surroundingScopeUnit0*.**s-implicit**∈*State-aggregation-node*)

∀*pn*∈*State-exit-point-name*: (*pn.parentAS1*∈*Inner-exit-point*) ⇒
(*pn*∈*parentAS1ofKind*(*pn, State-partition*).*baseType1.exitPointSet1*)

Likewise, the *Outer-exit-point*s must denote exit points in the inner and outer composite state, respectively.

∀*td*∈*Composite-state-type-definition*: (*td*.**s-implicit**∈*State-aggregation-node*)⇒
∃!*cd*∈*Connection-definition*: (*cd.surroundingScopeUnit0=td*)∧
(**let** *pointSet* = {*pn*∈*State-entry-point-definition*∪*State-exit-point-definition*:
(*pn*∈ *td.entryPointSet1*∪*td.exitPointSet1*)∨
(∃*sp*∈*State-partition*: (*sp*. *surroundingScopeUnit0= td*)∧
(*pn*∈*sp.baseType1.entryPointSet1*∪ *sp.baseType1.exitPointSet1*))} **in**
let *pointSet'*=
{*pn*∈*State-entry-point-definition*∪*State-exit-point-definition*: *isAncestorAS1*(*cd*,*pn*)} **in**
pointSet ⊆ *pointSet'*
endlet)

All entry and exit points of both the container state and the state partitions must appear in exactly one *Connection-definition*.

∀*sp*,*sp'*∈ *State-partition*: (*sp*≠*sp'*)∧ (*sp.parentAS1*= *sp'*. *parentAS1*)⇒
sp.inputSignalSet1⌒ *sp'*. *inputSignalSet1*=∅

The input signal sets of the *State-partition*s within a composite state must be disjoint.

Concrete syntax

<state aggregation> =
    <package use clause>* <state aggregation heading> <aggregation structure>

<state aggregation heading> =
    <composite state heading>

<aggregation structure> ::
    { <state connection points> | <entity in state aggregation> }*
    <state aggregation body>

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

<state partition connection> = <state partition connection entry>
  | <state partition connection exit>

<state partition connection entry> ::
    [<composite state<identifier>] <outer entry points>
    <composite state<identifier> <inner entry point>

<outer entry points> :: <state entry point>+ | **default**

<inner entry point> :: <nextstate parameters> | <state entry point> | **default**

<state entry point> = <state entry point<name>

<state partition connection exit> ::
    [<composite state<identifier>] <outer exit point>
    <composite state<identifier> <inner exit points>

<outer exit point> :: <state exit point> | **default**

<inner exit points> :: <state exit points> | **default**

<state partition> =
  <textual typebased state partition def> | <composite state reference> | <composite state>

Transformations

<aggregation structure>(*selist* = * ⌒ <state connection points>(* ⌒ < *n* > ⌒ *) ⌒ *,
    body = * ⌒ *s* ⌒ *)
provided *s* ∈ <state partition> ∧ *n* ∈ <state entry points> ∧
  < *i* **in** *body*: (*i* ∈ <state partition connection entry> ∧ *i*.**s-**<outer entry points>.**s-implicit** = *n* ∧
    i.**s2-**<identifier> = *s.identifier0*) > = *empty*
=8=>
  <aggregation structure>(*selist* ⌒
    <state partition connection entry>(*undefined*, <outer entry points>(< *n* >),
      s.identifier0, <inner entry point>(**default**)), *body*)

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

<aggregation structure>(*selist*, *body* = * ⌒ *s* ⌒ *)
provided *s* ∈ <state partition> ∧ *n* ∈.<state exit points> ∧ *n* ∈. *stateConnectionPointSet0* (*s*) ∧
  < *i* **in** *body*: (*i* ∈ <state partition connection exit> ∧ *i*.**s-**<inner exit points>.**s-implicit** = *n* ∧
    i.**s2-**identifier> = *s.identifier0*) > = *empty*
=8=>

<aggregation structure>(*selist* ⌒
<state partition connection exit>(*undefined*, <outer exit point>(**default**),
s.identifier0,<inner exit points>(< *n* >)), *body*)

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

The following statement is formalized by the dynamic semantics.

If there are signals in the complete valid input set of an agent that are not consumed by any state partition of a certain composite state, an additional implicit state partition is added to that composite state. This implicit partition has only an unlabelled start transition and a single state containing all implicit transitions (including those for exported procedures and exported variables). When one of the other partition exits, an implicit signal is sent to the agent, which is consumed by the implicit partition. After the implicit partition has consumed all the implicit signals, it exits through a State-return-node.

let *nn*=*newName* **in**
< <composite state>(*uses*, <composite state heading>(*, *n*, *params*), *struct*) >
=8=>
< <typebased composite state>(*n*, <type expression>(<identifier>(*undefined*, *nn*), *undefined*)),
<composite state type definition>(*uses*,
<composite state type heading>(*undefined*, *empty*, *nn*,
empty, *undefined*, *undefined*, *params*), *struct*) >

A State-definition has an implied anonymous state type that defines the properties of the state.

Auxiliary functions

Get the set of input signals appearing in a type definition, a state partition or a state node.

inputSignalSet1(*sp*: *TYPEDEFINITION1*∪*State-partition* ∪*State-node*): *SIGNAL1* =$_{\text{def}}$
if *sp*∈*State-node* **then**
sp.**s**-*Save-signalset*∪{*in*.**s**-*Signal-identifier*: *in*∈*sp*.**s**-*Input-node*-**set**}∪
{*getEntityDefinition1*(*cn*.**s**-*Procedure-identifier*,**procedure**).*inputSignalSet1*:
cn∈{*cn*∈*Call-node*: *isAncestorAS1*(*sp*, *cn*)}}
else // *sp*∈*TYPEDEFINITION1*∪*State-partition*
{*sn*.*inputSignalSet1*: *sn*∈*sp*.*stateNodeSet1*}
endif

Get the base type of a definition.

baseType1(*as*: *Agent-definition*∪*State-machine*∪*State-partition*∪*State-node*):
Agent-type-definition∪*Composite-state-type-definition*=$_{\text{def}}$
case *as* **of**
| *Agent-definition* => *getEntityDefinition1*(*as*.**s**-*Agent-type-identifier* **agent type**)
State-machine∪*State-partition*=>
getEntityDefinition1 (*as*.**s**-*Composite-state-type-identifier*, **state type**)
| *State-node* =>
if *as*.**s**-*Composite-state-type-identifier*=*undefined* **then** *undefined*
else *getEntityDefinition1* (*as*.**s**-*Composite-state-type-identifier*, **state type**)
otherwise *undefined*
endcase

Get the set of the state entry points of a *Composite-state-type-definition*.

entryPointSet1(*d*: *Composite-state-type-definition*): *Name*-**set**=$_{def}$
d.**s**-*State-entry-point-definition*-**set**

Get the set of the state exit points of a *Composite-state-type-definition*.

exitPointSet1(*d*: *Composite-state-type-definition*): *Name*-**set**=$_{def}$
d.**s**-*State-exit-point-definition*-**set**

Mapping to abstract syntax

| <state partition connection entry>(<outer entry points>(< *n1*>), <inner entry point>(*n2*))
=> **mk**-*Entry-connection-definition*
(**mk**-*Outer-entry-point*(*Mapping*(*n1*), **mk**-*Inner-entry-point*(*Mapping*(*n2*))

| <state partition connection exit>(<outer exit point>(*n1*), <inner exit points>(< *n2* >))
=> **mk**-*Exit-connection-definition*
(**mk**-*Outer-exit-point*(*Mapping*(*n1*), **mk**-*Inner-exit-point*(*Mapping*(*n2*))

F2.2.8.11.3 State connection point

Concrete syntax

<state connection points> = <state entry points> | <state exit points>

<state entry points> :: <state entry point>+

<state exit points> :: <state exit point>+

<state exit point> = <state exit point<name>

Mapping to abstract syntax

| <state entry points>(*x*) => *Mapping*(*x*)

| <state exit points>(*x*) => *Mapping*(*x*)

F2.2.8.11.4 Connect

Abstract syntax

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

Concrete syntax

<connect part> ::
[<virtuality>] <connect list> <exit transition>

<exit transition> = <transition>

<connect list> = <state exit point list> | <asterisk connect list>

<state exit point list> = <state exit point<name>*

<asterisk connect list> :: ()

Conditions on concrete syntax

$\forall cl \in$ <connect list>: $\forall pn \in$ <name>:
isAncestorAS0(*cl*,*pn*) $\Rightarrow$
($\exists scp \in$ <state connection points>: $\exists sep \in$ <state exit points>:
isAncestorAS0 (*scp*, *sep*) $\wedge$ *isAncestorAS0* (*scp.parentAS0*, *pn*) $\wedge$ (*pn* = *sep*))

The <connect list> must only refer to visible <state exit point>s.

Transformations

< <connect part>(*virt*, < *n* > $\frown$ *rest*, *trans*) > **provided** *rest* $\neq$ *empty*
=1=> < <connect part>(*virt*, < *n* >, *trans*), <connect part>(*virt*, *rest*, *trans*) >

When the <connect list> of a certain <connect part> contains more than one <state exit point>, a copy of the <connect part> is created for each such <state exit point>. Then the <connect part> is replaced by these copies.

c=<connect part>(*virt*, <asterisk>, *trans*)
=13=>
(**let** *parentType* = *c*.*parentAS0*.**s-**<state list>.*head*.**s-**<type expression>.*refersto0* **in**
let *allExits* = *bigSeq*(< < *ex*.**s-**<name> | *ex* **in** *exits* > |
exits **in** *parentType*.**s-**<composite state structure>.**s-**<state connection points>**-seq**:
(*exits* ∈ <state exit points>) >) **in**
<connect part>(*virt*, *allExits*, *trans*)
endlet
endlet)

A <connect list> that contains an <asterisk connect list> is transformed into a list of <state exit point>s, one for each <state exit point> of the <composite state> in question. The list of <state exit point>s is then transformed as described above.

Mapping to abstract syntax

| <connect part>(*, < *name* >, *trans*) =>
mk-*Connect-node*(*Mapping*(*name*),*Mapping*(*trans*))

F2.2.8.12 Transition

F2.2.8.12.1 Transition body

Abstract syntax

Transition :: *Graph-node** { *Terminator* | *Decision-node*}

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

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

Concrete syntax

<transition> = <transition action items> | <terminator>

<transition action items> :: <transition string> [<terminator>]

<transition string> = <action>+

<action> :: [<label>] <action item>

<action item> =
<task>
| <output>
| <create request>
| <decision>
| <set>
| <reset>
| <export statement>

| <procedure call>
| <remote procedure call>
| <transition option>

<terminator> :: [<label>] <terminator node>

<terminator node> = <nextstate> | <join> | <stop> | <return>

Conditions on concrete syntax

∀*t*∈<transition>: (*t*∉<terminator>)∧(*t*.**s-**<terminator> = *undefined*)∧
(*t.parentAS0.parentAS0.parentAS0*∉<decision>∪<transition option>) ⇒
(**let** *asl* = *t*.**s-**<action>**-seq in**
(*asl.last*.**s-**<action item>∈<decision>∪<transition option>)∧
isTransitionTerminating0(*asl.last*.<action item>) **endlet**)

endlet)

If the <terminator> of a <transition> is omitted, then the last action in the <transition> must contain a terminating <decision> or terminating <transition option>, except when a <transition> is contained in a <decision> or <transition option>.

Transformations

t=<terminator>(*,*) **provided** *t.parentAS0* ∉ <transition>
=1=> <transition action items>(*empty*, *t*)

This rule unifies the two possible representations for <transition> into one. Please note that the resulting structure would not be valid concrete syntax. However, this is remedied by the transformations for decisions.

The following transformation is handled in the transformations for remote procedure call and import expression.

A transition action may be transformed to a list of actions (possibly containing implicit states) according to the transformation rules for <import expression> and <remote procedure call>.

Mapping to abstract syntax

| <transition action items>(*s*, *t*)
=> **if** *t* = *undefined* **then mk-***Transition*(*Mapping*(< *x* **in** *s*: (*x* ∉ <decision>) >), *Mapping*(*s.last*))
else mk-*Transition*(*Mapping*(*s*), *Mapping*(*t*))
endif

| <action>(*, *a*) => *Mapping*(*a*)

Auxiliary functions

Determine if a <decision> or a <transition option> is terminating.

isTransitionTerminating0(*dt*: <decision>∪<transition option>): *BOOLEAN* =$_{\text{def}}$
∀*t*∈<transition>: (*t.parentAS0.parentAS0.parentAS0*= *dt*) ⇒
((*t*∈<terminator>)∨
((*t*∉<terminator>)∧(*t*.**s-**<terminator> ≠ *undefined*))∨
((*t*∉<terminator>)∧(*t*.**s-**<terminator> = *undefined*)∧
isTransitionTerminating0(*t*.**s-**<action>**-seq**.*last*.<action item>))

F2.2.8.12.2 Nextstate

Abstract syntax

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

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

Conditions on abstract syntax

∀*nn* ∈ *Nextstate-node*: *nn*.**s**-*Nextstate-parameters* ≠ *undefined*⇒
((*nn*.**s**-*State-name* = *sn*.**s**-*State-name*)∧
(*parentAS1ofKind*(*nn*, *State-transition-graph*∪*Procedure-graph*) =
parentAS1ofKind(*sn*, *State-transition-graph*∪*Procedure-graph*))∧
(*nn*.**s**-*Nextstate-parameters* ≠ *undefined*))⇒
sn.**s**-*Composite-state-type-identifier*≠ *undefined*)

The *State-name* specified in a nextstate must be the name of a state within the same *State-transition-graph* or *Procedure-graph*. If a *Named-nextstate* includes *Nextstate-parameters*, the *State-name* shall refer to a composite state (a *State-node* with a *Composite-state-type-identifier*).

Concrete syntax

<nextstate> = <nextstate body>

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

<nextstate body gen name> ::
<state><name> <nextstate parameters>

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

<dash nextstate> :: ()

<history dash nextstate> :: ()

Conditions on concrete syntax

∀*s*∈<state>: ∀*t*∈<transition>:
(*t.parentAS0*∈<input part>∪<priority input>∪<spontaneous transition>∪<continuous signal>) ∧
(*t.parentAS0.parentAS0*= *s*) ∧
(∃*hdn*∈<history dash nextstate>: *isAncestorAS0*(*t*,*hdn*)) ⇒
(*s.surroundingScopeUnit0*∈<composite state> ∪<composite state type definition>)

If a transition is terminated by a <history dash nextstate>, the <state> must be a <composite state>.

∀*t*∈<transition>: (*t.parentAS0*∈<start>) ⇒ ¬(∃*dn*∈<dash nextstate>: *isAncestorAS0*(*t*,*dn*))

The <transition> contained in a <start> must not lead, directly or indirectly, to a <dash nextstate>.

∀*t*∈<transition>: (*t.parentAS0*∈<start>) ⇒
¬(∃*dn*∈<history dash nextstate>: *isAncestorAS0*(*t*,*dn*))

The <transition> contained in a <start> must not lead, directly or indirectly, to a <history dash nextstate>.

Transformations

The following text is handled by the dynamic semantics.

In each <nextstate> of a <state> the <dash nextstate> is replaced by the <state name> of the <state>. This model is applied after the transformation of <state>s and all other transformations except those for trailing commas, synonyms, priority inputs, continuous signals, enabling conditions, implicit tasks for imperative actions and remote variables or procedures.

F2.2.8.12.3 Join

Abstract syntax

Join-node :: *Connector-name*

Concrete syntax

<join> :: <connector<name>

Conditions on concrete syntax

∀*b*∈<agent body>∪<procedure body>∪<operation body>∪<composite state body>:
∀*j*∈<join>: *isAncestorAS0*(*b*, *j*) ⇒
(∃!*l*∈<label>: *isAncestorAS0*(*b*, *l*)∧(*j*.**s**-<name> = *l*.**s**-<name>))

There must be exactly one <connector<name> corresponding to a <join> within the same body.

Mapping to abstract syntax

| <join>(*name*) => **mk**-*Terminator*(**mk**-*Join-node*(*Mapping*(*name*)), *undefined*)

F2.2.8.12.4 Stop

Abstract syntax

Stop-node :: ()

Conditions on abstract syntax

Concrete syntax

<stop>:: ()

Mapping to abstract syntax

| <stop>() => **mk**-*Terminator*(**mk**-*Stop-node*(), *undefined*)

F2.2.8.12.5 Return

Abstract syntax

Return-node	=	*Action-return-node*
	|	*Value-return-node*
	|	*Named-return-node*
Action-return-node	::	()
Value-return-node	::	*Expression*
Named-return-node	::	*State-exit-point-name*

Conditions on abstract syntax

∀*rn* ∈ *Return-node*: ∃*pg* ∈ *Procedure-graph*: *isAncestorAS1*(*pg*, *rn*)

A *Return-node* must be contained in a *Procedure-graph*.

∀*rn* ∈ *Action-return-node*: ∃*d* ∈ *Procedure-definition*:
isAncestorAS1(*d*.**s**-*Procedure-graph*, *rn*) ∧ *d*.**s**-*Result* = *undefined*

An *Action-return-node* must only be contained in the *Procedure-graph* of a *Procedure-definition* without *Result*.

∀*rn* ∈ *Value-return-node*: ∃*d* ∈ *Procedure-definition*:
isAncestorAS1(*d*.**s**-*Procedure-graph*, *rn*) ∧ *d*.**s**-*Result* ≠ *undefined*

A *Value-return-node* must only be contained in the *Procedure-graph* of a *Procedure-definition* containing *Result*.

∀*rn* ∈ *Named-return-node*: ∃*sg* ∈ *Composite-state-graph*: *isAncestorAS1*(*sg*, *rn*)

A *Named-return-node* must only be contained in a *Composite-state-graph*.

Concrete syntax

<return> ::<return body> | <state exit point<name>

<return body> :: [<expression>]

Mapping to abstract syntax

| <return>(*x*)
  => **if** *x* = *undefined* **then mk-***Terminator*(**mk-***Action-return-node*())
    elseif *x* ∈ <name> **then mk-***Terminator*(**mk-***Named-return-node*(*Mapping*(*x*)))
    else mk-*Terminator*(**mk-***Value-return-node*(*Mapping*(*x*)
    endif

F2.2.8.13 Action

F2.2.8.13.1 Task

Abstract syntax

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

Concrete syntax

<task> :: { <task body> | <informal text> | <legacy task body> }

<task body> :: <variable definitions> <statements>

<legacy task body> = <assignment>+

Transformations

< <task>(<task body>(<statements>(*, *empty*))) >
=1=> *empty*

If the <statements> in the <task body> of <task> is empty, then the <task> is removed. Any syntactic item leading to the <task> or <task area> shall then lead directly to the item following the <task> or <task area>, respectively.

< <task>(<task body>(*vl*, *sl*)) >
=1=> < <compound statement>(*undefined*, *vl*, *sl*) >

A <task> containing a <task body> is transformed into a <compound statement>, and then the <compound statement> is transformed as shown in clause F2.2.8.14.1. The result of this transformation is inserted in place of <task>.

Further study is needed to verify the formalization of the transformation of a <task body>.

< <task>(<legacy task body>(*al*)) >
=1=> < <compound statement>(*undefined*, *undefined*, *al*) >

A <task> containing a <legacy task body> is transformed into a <compound statement> that is a list of assignments, and then the <compound statement> is transformed as shown in clause F2.2.8.14.1. The result of this transformation is inserted in place of <task>.

NOTE – The transform of a <task> or <task area> containing a <task body> is not necessarily mapped onto a Task-node in the Abstract Grammar.

Mapping to abstract syntax

| <task>(*t*) => **mk-***Graph-node*(*Mapping*(*t*))

F2.2.8.13.2 Create

Abstract syntax

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

Conditions on abstract syntax

∀*n* ∈ *Create-request-node*: ∀*d* ∈ *Agent-definition*:
(*d*=*getEntityDefinition1*(*n*.**s**-*Agent-identifier*,**agent**))⇒
(∃*t* ∈ *Agent-type-definition*:
(*t*=*getEntityDefinition1*(*d*. **s**-*Agent-type-identifier*,**agent type**)) ∧
isActualAndFormalParameterMatched1(*n*.**s**-*Expression*-**seq**, *t*.*formalParameterSortList1*))

The length of the list of *Actual-parameters* must be the same as the number of *Agent-formal-parameter*s in the *Agent-definition* of the *Agent-identifier* and each *Expression* of *Actual-parameter*s corresponding by position to an *Agent-formal-parameter* must have a sort that is compatible to the sort of the *Agent-formal-parameter* in the *Agent-definition* denoted by *Agent-identifier*.

Concrete syntax

<create request> :: <create body>

<create body> :: { <identifier> | **this** } <actual parameters>

Conditions on concrete syntax

∀*cr*∈<create body>: (*cr*.**s-implicit** = **this**) ⇒
(*cr*.*surroundingScopeUnit0*∈<agent type definition>) ∧
(*cr*.*surroundingScopeUnit0*.*surroundingScopeUnit0*∈<agent type definition>)

this may only be specified in an <agent type definition> and in scopes enclosed by an <agent type definition>.

Transformations

The following statement is formalized in the dynamic semantics.

Stating **this** is derived syntax for the implicit <process identifier> that identifies the set of instances of the agent in which the create is being interpreted.

c=<create request>(<create body>(*id*, *params*))
provided *id*.*refersto0* ∈ <agent type definition> ∧ *c*.*possibleInstances0* ≠ *empty*
=8=>
(**let** *inst* = *c*.*possibleInstances0* **in**
if *inst*.*length* = 1 **then** <create request>(<create body>(*inst*.*head*.*identifier0*, *params*))
else <decision>(**any**, <textual decision body>(<
<textual answer part>(*undefined*,
<transition action items>(<action>(*undefined*,
<create request>(<create body>(*elem*.*identifier0*, *params*)),
undefined))
| *elem* **in** *inst* >, *undefined*))
endif
endlet)

If <agent type identifier> is used in a <create request> and there exists one instance set of the indicated agent type in the agent containing the instance that performs the create, the <agent type identifier> is derived syntax denoting this instance set.

If there is more than one instance set it is determined at interpretation time in which set the instance will be created. The <create request> is in this case replaced by a non-deterministic decision using any followed by one branch for each instance set. In each of the branches a create request for the corresponding instance set is inserted.

let *nn* = *newName* **in**
c=<create request>(<create body>(*id*, *params*)
provided *id*.*refersto0* ∈ <agent type definition> ∧ *c*.*possibleInstances0* = *empty*
=8=>
<create request>(<create body>(<identifier>(*undefined*, *nn*), *params*)
and

entities = *parentAS0ofKind*(*c*, <agent definition> ∪ <agent type definition>).*getEntities*
=>
entities ⁀
if *id*.*refersto0* ∈ <system type definition>
then <textual typebased system definition>(
<typebased system heading>(*nn*, <type expression>(*id*, *empty*)))
elseif *id*.*refersto0* ∈ <block type definition>
then <textual typebased block definition>(
<typebased block heading>(*nn*, <number of instances>(*undefined*, *undefined*),
<type expression>(*id*, *empty*)))
else *id*.*refersto0* ∈ <process type definition>
then <textual typebased process definition>(*nn*, <number of instances>(*undefined*, *undefined*),
<type expression>(*id*, *empty*))
endif

If there does not exist any instance set of the indicated agent type in the containing agent then:

a) an implicit instance set of the given type with a unique name is created in the containing agent; and

b) the <agent identifier> in the <create request> is derived syntax for this implicit instance set.

Auxiliary functions

The following function aims at finding the possible instances for an agent type create request.

possibleInstances0(*c*: <create request>): <agent definition>* =$_{def}$
< *e* **in** *parentAS0ofKind*(*c*, <agent definition> ∪ <agent type definition>).*getEntities*:
e ∈ <agent definition> ∧ *e*.**s-**<type expression>.**s-**<base type> = *c*.**s-**<create body>.**s-implicit** >

Mapping to abstract syntax

| <create request>(*c* => **mk-***Graph-node*(*Mapping*(*c*))

| <create body>(*id*, *params*) => **mk-***Create-request-node*(*Mapping*(*id*), *Mapping*(*params*))

F2.2.8.13.3 Procedure call

Abstract syntax

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

Conditions on abstract syntax

∀*n* ∈ *Call-node* ∪ *Value-returning-call-node*: ∀*d* ∈ *Procedure-definition*:
(*d*=*getEntityDefinition1*(*n*. **s**-*Procedure-identifier*,**procedure**)) ⇒
(*isActualAndFormalParameterMatched1*(*n*.**s**-*Expression*-**seq**, *d*.*formalParameterSortList1*)∧
(∀*i* ∈ 1..*n*.**s**-*Expression*-**seq**.*length*:
d. *formalParameterList1*[*i*] ∈ *Inout-parameter* ∪ *Out-parameter*⇒
n.**s**-*Expression*[*i*]∈ *Identifier*∧*n*.**s**-*Expression*[*i*].*idKind1*= **variable**))

The length of the list of optional *Expression*s must be the same as the number of the *Procedure-formal-parameter*s in the *Procedure-definition* denoted by the *Procedure-identifier* and each *Expression* corresponding by position to an *In-parameter* must be sort compatible to the sort of the *Procedure-formal-parameter*. Each *Expression* corresponding by position to an *Inout-parameter* or *Out-parameter* must be a *Variable-identifier* that is sort compatible to the sort identified by the *Sort-reference-identifier* of the *Procedure-formal-parameter*.

Concrete syntax

<procedure call> :: <procedure call body>

<procedure call body> ::
[**this**] {procedure<identifier> | procedure<type expression>} <actual parameters>

Conditions on concrete syntax

∀*pc*∈<procedure call>:
(**let** *apl* = *pc*.**s-**<procedure call body>.**s-**<actual parameter>**-seq in**
let *fpl* = *pc.calledProcedure0.procedureFormalParameterList0* **in**
(*fpl.length* = *apl.length*) ∧
(∀*i*∈1..*fpl.length*:
(*fpl*[*i*].*parentAS0.parentAS0*.**s-**<parameter kind>∈{**inout**, **out**}) ⇒
(*apl*[*i*] ≠ *undefined*) ∧ (*apl*[*i*]∈<variable access>∪<extended primary>)) **endlet**)))

An <expression> in <actual parameters> corresponding to a formal **in**/**out** or **out** parameter cannot be omitted and must be a <variable access> or <extended primary>.

∀*pcd*∈<procedure call body>: (*pcd*.**s-this** ≠ *undefined*) ⇒
parentAS0ofKind(*pcd*, <internal procedure definition>)= *getEntityDefinition0*(*pcd*.**s-**<identifier>, **procedure**)

If **this** is used, <procedure identifier> must denote an enclosing procedure.

Transformations

let *nn*= *newName* **in**
p=<procedure call body>(*id*, *params*)
provided *parentAS0ofKind*(*id.refersto0*, <agent type definition>) ≠
parentAS0ofKind(*p*, <agent type definition>)
=8=>
let *par*=*parentAS0ofKind*(*p*, <agent type definition>) **in**
<procedure call body>(
<identifier>(*par.fullQualifier0*⌒<path item>(*par.entityKind0*, *par.entityName0*), *nn*),
params)
endlet
and // add the new definition
let *defs*=
parentAS0ofKind(*p*, <agent type definition>).**s-**<agent structure>.**s-**<entity in agent>**-seq in**
defs => *defs* ⌒
<internal procedure definition>(*empty*,
<procedure heading>(
<procedure preamble>(*undefined*, *undefined*),
empty, *nn*, *empty*, *undefined*, *undefined*, *empty*, *undefined*, *empty*),
empty,
<procedure body>(*undefined*, *undefined*, *empty*))
endlet

If the <procedure identifier> is not defined within the enclosing agent, the procedure call is transformed into a call of a local, implicitly created subtype of the procedure.

The following statement is handled by the dynamic semantics.

The keyword **this** implies that when the procedure is specialized, the <procedure identifier> is replaced by the identifier of the specialized procedure.

Mapping to abstract syntax

| <procedure call>(*p* => **mk-***Graph-node*(*Mapping*(*p*))

| <procedure call body>(*t*, *id*, *params*) =>
mk-*Call-node*(*Mapping*(*t*), *Mapping*(*id*), *Mapping*(*params*))

F2.2.8.13.4 Output

Abstract syntax

Output-node	::	*Signal-identifier*
		Actual-parameters
		Activation-delay

		Signal-priority [*Signal-destination*] *Direct-via*
Activation-delay	=	*Expression*
Signal-priority	=	*Expression*
Signal-destination	= | |	*Expression* *Agent-identifier* **THIS**
Direct-via	=	*Gate-identifier*-**set**

Conditions on abstract syntax

∀*n* ∈ *Output-node*: ∃*d* ∈ *Signal-definition*:
(*d*=*getEntityDefinition1*(*n*.**s**-*Signal-identifier*,**signal**)) ∧
isActualAndFormalParameterMatched1(*n*.**s**-*Expression*-**seq**, *d*.*formalParameterSortList1*)

For each *Gate-identifier* in *Direct-via*, there shall exist zero or more channels such that the gate via this path is reachable with the *Signal-identifier* from the agent and the *Out-signal-identifier*-***set*** of the gate shall include the *Signal-identifier*.

The sort of *Expression* of a *Signal-destination* shall either be the sort Pid (see clause 12.1.5), or the sort *Interface-definition* with the *Signal-identifier* in its *Signal-identifier*-***set***.

The *Agent-identifier* of a *Signal-destination* shall identify an agent reachable from the originating agent.

The length of the list of optional *Expression*s must be the same as the number of *Sort-reference-identifier*s in the *Signal-definition* denoted by the *Signal-identifier*. Each *Expression* must be sort compatible to the corresponding (by position) *Sort-identifier-reference* in the *Signal-definition.*

Concrete syntax

<output> :: <output body>

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

<output body item> ::
{ <signal<identifier> [<actual parameters>] | <expression output> | <encoded output> }
[<activation delay>][<signal priority>]

<expression output> :: <expression>

<encoded output> :: <expression>

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

<destination> :: { <pid<expression> | <agent<identifier> | **this** }

<activation delay> :: < Duration<expression>

<signal priority> :: <Natural<expression>

Further study is needed for <expression output> and <encoded output>.

Conditions on concrete syntax

∀*op*∈<output>: ∀*dt*∈<destination>:
(*dt*.*parentAS0* = *op*.**s**-<output body>) ∧ (*dt*.**s**-**implicit** = **this**) ⇒
(*op*.*surroundingScopeUnit0*∈<agent type definition>) ∧
(*op*.*surroundingScopeUnit0*.*surroundingScopeUnit0*∈<agent type definition>)

this may only be specified in an <agent type definition> and in scopes enclosed by an <agent type definition>.

∀*op*∈<output>:∀*dt*∈<destination>:

(*dt.parentAS0* = *op*.**s-**<output body>) ∧
(*dt*.**s-implicit** ∈<expression>) ∧ (*dt.staticSort0* ≠ "Pid") ⇒
(**let** *fd* = *getEntityDefinition0*(*dt.staticSort0*, **interface**) **in**
∀*sig*∈<identifier>: (*sig.parentAS0* = *op*.**s-**<output body>) ⇒
(*sig*∈*fd.usedSignalSet0*) ∨ (*getEntityDefinition0*(*sig*, **signal**)∈*fd.definedSignalSet0*)
endlet)

If <destination> is a <pid expression> with a static sort other than Pid, the <signal identifier> must represent a signal defined or used by the interface that defined the pid sort.

Transformations

< <output>(<output body>(< *o* > ⌒ *rest*, *constr*)) > =1=>
< <output>(<output body>(< *o* >, *constr*)), <output>(<output body>(*rest*, *constr*)) >

If several pairs of <signal identifier> and <actual parameters> are specified in an <output body>, this is derived syntax for specifying a sequence of <output>s or <output area>s in the same order as specified in the original <output body>, each containing a single pair of <signal identifier> and <actual parameters>. The to <destination> clause and the <via path>s are repeated in each of the <output>s or <output area>s.

Further study is needed for the transformation of an <output body> with a <communication constraints> with more than one <destination>, <activation delay> or <signal priority>.

The following statement is covered by the dynamic semantics.

Stating **this** in <destination> is derived syntax for the implicit <agent identifier> that identifies the set of instances for the agent in which the output is being interpreted.

Mapping to abstract syntax

| <output>(*o*) => **mk-***Graph-node*(*Mapping*(*o*)
| <output body>(< <output body item>(*id*, *params*, *delay*, *priority*) >, *constr*)
=> **mk-***Output-node*(*Mapping*(*id*), *Mapping*(*params*), *Mapping*(*delay*), *Mapping*(*priority*),
Mapping(*head*(< *c* **in** *constr*: (*c* ∈ <destination>) >)),
Mapping(< *c* **in** *constr*: (*c* ∈ <via path>) >).*toSet*)
| <destination>(*d*) => *Mapping*(*d*)
| <via path>(*gate_id*) =>*Mapping*(*gate_id*)
| <activation delay>(*delay*) =>
if *delay=undefined*
then <Duration<expression>(*0*)
else <Duration<expression>(*delay*)
endif
| <signal priority>(*priority*) =>
if *priority=undefined*
then <Natural<expression>(*0*)
else < Natural<expression>(*priority*)
endif

Auxiliary functions

usedSignalSet0(*fd*: <interface definition>): *SIGNAL0* =$_{\text{def}}$
{*sig*∈*SIGNAL0*:
(*sig*∈*fd*.**s-**<entity in interface>.**s-**<interface use list>.**s-**<signal list item>**-seq**.*signalSet0*) ∨
(∃*fd'*∈<interface definition>: *isSubtype0*(*fd*, *fd'*) ∧
(*sig*∈*fd'*.**s-**<entity in interface>.**s-**<interface use list>.**s-**<signal list item>**-seq**.*signalSet0*))}

definedSignalSet0(*fd*: <interface definition>): <signal definition>**-set** =$_{\text{def}}$
{*sd*∈<signal definition>:
((*sd.parentAS0*∈<signal definition list>)∧(*sd.parentAS0.parentAS0*∈<entity in interface>) ∧
(*sd.parentAS0.parentAS0.parentAS0* = *fd*)) ∨
(∃*fd'*∈<interface definition>: *isSubtype0*(*fd*, *fd'*) ∧ (*sd.parentAS0*∈<signal definition list>) ∧

(*sd.parentAS0.parentAS0*∈<entity in interface>) ∧
(*sd.parentAS0.parentAS0.parentAS0*= *fd'*))}

F2.2.8.13.5 Decision

Abstract syntax

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

Conditions on abstract syntax

∀*dn* ∈ *Decision-question*: ∀*r, r'* ∈ *Range-condition*: ∀*ce,ce'* ∈ *Constant-expression*:
isAncestorAS1(*r,ce*)∧ *isAncestorAS1*(*r',ce'*) ∧ (*ce*≠*ce'*)∧
r'. parentAS1∈ *dn*.**s**-*Decision-answer*-**set** ∧ *r. parentAS1*∈ *dn*.**s**-*Decision-answer*-**set**⇒
(*isCompatibleTo1*(*ce. staticSort1, ce'. staticSort1*)∨
isCompatibleTo1(*ce'. staticSort1, ce. staticSort1*)) ∧
(*dn*.**s**-*Decision-question*∈*Expression*⇒
isCompatibleTo1(*ce. staticSort1, dn*.**s**-*Decision-question. staticSort1*))

If the *Decision-question* is an *Expression*, the *Range-condition* of the *Decision-answer*s must be sort compatible to the sort of the *Decision-question*. The *Constant-expression*s of the *Range-condition*s must be of a compatible sort.

∀*dn* ∈*Range-condition*: ∀*r, r'* ∈ *Range-condition*:
r'. parentAS1 ∈ *dn*.**s**-*Decision-answer*-**set** ∧ *r. parentAS1*∈ *dn*.**s**-*Decision-answer*-**set** ∧
r. parentAS1≠ *r'. parentAS1* ⇒ *r.range1* ⌒ *r'. range1* = ∅

The *Range-condition*s of the *Decision-answer*s must be mutually exclusive.

Concrete syntax

<decision> :: <question> <textual decision body>

<textual decision body> :: <textual answer part>+ [<textual else part>]

<textual answer part> :: [<answer>] [<transition>]

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

<textual else part> :: [<transition>]

<question> = <expression> | <informal text> | **any**

Conditions on concrete syntax

∀*d*∈<decision>: (*d*.**s**-<question> = **any**) ⇔
¬(∃*ap*∈<textual answer part>: (*ap.parentAS0.parentAS0*= *d*)) ∧ (*ap*.**s**-<answer> ≠ *undefined*) ∧
(*d*.**s**-<textual decision body>.**s**-<textual else part> = *undefined*)

The <answer> of <textual answer part> shall be omitted if and only if the <question> consists of the keyword **any**. In this case, a <textual else part> shall be absent.

Transformations

<textual else part>(*undefined*) =1=>
<textual else part>(<transition action items>(*empty*, *undefined*))

<textual answer part>(*a*, *undefined*) =1=>
<textual answer part>(*a*, <transition action items>(*empty*, *undefined*))

These first two transformations are used to insert an empty transition instead of an undefined one. This empty transition will be filled with a terminator within the step below (inserting terminating actions into the transition).

t=<transition action items>(*a*, *undefined*)
provided *a.last* ∉ <decision> ∧ *t.parentAS0.parentAS0.parentAS0* ∈ <decision> ∧
t.findContinueLabel ≠ *undefined*
=5=> <transition action items>(*a*,
<terminator>(*undefined*,<join>(*findContinueLabel*(*t*))))

If a <decision> is not terminating, it is derived syntax for a <decision> where all not terminating <textual answer part>s and the <textual else part> (if not terminating) have inserted at the end of their <transition> a <join> to the first <action> following the decision or (if the decision is the last <action> in a <transition string>) to the following <terminator>.

< *d* =<decision>(*, *), <action>(*undefined*, *a*) > **provided** ¬ *terminatingDecision*(*d*)
=5=> < *d*, <action>(*newName*, *a*) >

<transition action items>(*str*, <terminator>(*undefined*, *t*))
provided *str.last* ∈ <decision> ∧ ¬ *str.last.terminatingDecision*
=5=> <transition action items>(*str*, <terminator>(*newName*, *t*))

The rules above insert a new label after a non-terminating decision.

let *nn* = *newName* **in**
d=<decision>(**any**, <textual decision body> (*ans*, *undefined*))
=8=>
<decision>(<operand5>(*undefined*, <any expression>(<identifier>(*empty*, *nn*))),
<textual decision body>(
< <textual answer part>(<operand5>(*undefined*, *idx*), *ans*[*idx*].s-<transition>)
| *idx* **in** 1 .. *ans.length* >,
undefined))
and
let *parent* = *d.surroundingScopeUnit0* **in**
entities=parent.getEntities
=>
entities ⌒ < <syntype definition>(*empty*,
<syntype definition syntype>(*nn*,
<identifier>(< <path item>(**package**, "Predefined") >, "Integer"),
undefined,
< <closed range>("1", *ans.length*) >)) >

Using only **any** in a <decision> is shorthand for using <any expression> in the decision. Assuming that the <textual decision body> is followed by N <textual answer part>s, any in <decision> is then shorthand for writing **any**(data_type_N), where data_type_N is an anonymous syntype defined as

syntype data_type_N =
package Predefined Integer **constants** 1:N
endsyntype;

The omitted <answer> items are shorthands for writing the literals 1 through N as the <constant> items of the <range condition> items in the N <answer> items.

Mapping to abstract syntax

| <decision>(*q*, <textual decision body>(*answers*, *elseAnswer*))
=> **mk-***Decision-node*(*Mapping*(*q*), *Mapping*(*answers*).*toSet*, *Mapping*(*elseAnswer*))

| <textual answer part>(*ans*, *trans*)

=> **mk-***Decision-answer*(*Mapping*(*ans*), *Mapping*(*trans*))

| <textual else part>(*trans*)
=> **if** *trans=undefined* **then** *undefined* **else mk-***Else-answer*(*Mapping*(*trans*)) **endif**

Auxiliary functions

rangeConditionList0(*d*:<decision>):<range condition>*=$_{def}$
let *apl* = *d*.**s-**<textual decision body>.**s-**<textual answer part> **in**
apl.answerPartRangeConditionList0
endlet

answerPartRangeConditionList0(*apl*:<textual answer part>*):<range condition>*=$_{def}$
if *apl.head*.**s-**<answer>∈<range condition> **then**
apl.head.**s-**<answer>⁀*apl.tail*. *answerPartRangeConditionList0*
else *apl.tail*. *answerPartRangeConditionList0*

The function *findContinueLabel* computes the continuation label after a decision within a transition string.

findContinueLabel(*x*: *DefinitionAS0*): <name> =$_{def}$
if *x* ∈ <transition action items> ∧ *x*.**s-**<terminator> ≠ *undefined* ∧
x.**s-**<terminator>.**s-**<label> = *undefined* ∧
x.**s-**<terminator>.**s-**<terminator node> ∈ <join>
then *x*.**s-**<terminator>.**s-**<terminator node>.**s-**<name>
else *findContinueLabel*(*x.parentAS0*)
endif

terminatingDecision(*d*: <decision>): *BOOLEAN* =$_{def}$
(∀ *a* ∈ *d*.**s-**<textual answer part>: *terminatingTransition*(*a*.**s-**<transition>)) ∧
(*d*.**s-**<textual else part> = *undefined* ∨
terminatingTransition(*d*.**s-**<textual else part>.**s-**<transition>))

A <decision> is a terminating decision, if each <textual answer part> and <textual else part> in its <textual decision body> is a terminating <textual answer part> or <textual else part> respectively.

terminatingTransition(*t*: <transition>): *BOOLEAN* =$_{def}$
t ∈ <terminator> ∨
t.**s-**<terminator node> ≠ *undefined* ∨
(**let** *d*= *t*.**s-**<action>.*last* **in** *d* ∈ <decision> ∧ *terminatingDecision* (*d*) **endlet**)

A <textual answer part> or <textual else part> in a decision is a terminating <textual answer part> or <textual else part> respectively if it contains a <transition> where a <terminator> is specified, or contains a <transition string> whose last <action> contains a terminating decision.

F2.2.8.14 Statement lists

Concrete syntax

<statements> =
<non terminating statements> [[<connector name>] <terminating statement>]
| [<connector name>] <terminating statement>

<non terminating statements> = <non terminating statement>+

<non terminating statement> =
[<connector name>] <statement>
| <compound statement>
| <loop statement>
| <decision statement>

<statement> =
<assignment statement>
| <set statement>

| <reset statement>
| <output statement>
| <create statement>
| <export statement>
| <call statement>
| <expression statement>

<set statement> = <set body>

<reset statement> = <reset body>

<output statement> = <output body>

<create statement> = <create body>

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

<variable definitions> = <variable definition statement>*

<variable definition statement> :: <local variables of sort>+

<local variables of sort> :: <aggregation kind> <variable<name>+ <sort> [<expression>]

Transformations

< <variable definition statement>(< *v* > ⌒ *rest*) > **provided** *rest* ≠ *empty* =1=>
< <variable definition statement>(< *v* >), <variable definition statement>(*rest*) >

A <variable definition statement> may contain several <local variables of sort>s. This is derived syntax for specifying a sequence of <variable definition statement>s, one for each <local variables of sort>. This is an auxiliary transformation.

< <local variables of sort>(*ak*, < *v* > ⌒ *rest*, *s*, *expr*) > **provided** *rest* ≠ *empty* =1=>
< <local variables of sort>(*ak*, < *v* >, *s*, *expr*), <local variables of sort>(*ak*, *rest*, *s*, *expr*) >

A <local variables of sort> may contain several <variable<name>s. This is derived syntax for specifying a sequence of <local variables of sort>s, one for each <variable<name>. This is an auxiliary transformation.

Mapping to abstract syntax

| <variable definition statement>(< *var* >) => *Mapping*(*var*)

| <local variables of sort>(*ak*, < *var* >, *s*, *expr*)
=> **mk-***Variable-definition*(*Mapping*(*var*), *Mapping*(*s*), *Mapping*(*ak*), *Mapping*(*expr*))

F2.2.8.14.1 Compound statement

Abstract syntax

Compound-node	::	*Connector-name* *Variable-definition*-**set** *Init-graph-node** *While-graph-node* *Transition* *Step-graph-node**
Init-graph-node	=	*Graph-node*
While-graph-node	=	*Expression** [*Finalization-node*]
Finalization-node	=	*Graph-node*
Step-graph-node	=	*Graph-node*

Continue-node :: *Connector-name*

Break-node :: *Connector-name*

Concrete syntax

<compound statement> :: [<connector name>] [<variable definitions>] <statements>

Conditions on abstract syntax

$\forall cn \in$ *Continue-node*: $\exists comp \in$ (*Compound-node*) :
isAncestorAS1(*comp, cn*) $\wedge$ (*cn*.**s**-*Connector-name* = *comp*.**s**-*Connector-name*)

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

$\forall bn \in$ *Break-node*: $\exists comp \in$ (*Compound-node*) :
isAncestorAS1(*comp, bn*) $\wedge$ (*bn*.**s**-*Connector-name* = *comp*.**s**-*Connector-name*)

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

Transformations

<compound statement>(*undefined*,*vardefs*,*stmts*)
=1=> <compound statement>(*newName*,*vardefs*,*stmts*)

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

The following statements are handled by the Mapping.

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

If the <compound statement> contains <variable definitions>, the following is performed for each <variable definition statement>. A new <variable name> is created for each <variable name> in the <variable definition statement>. Each occurrence of <variable name> in the following <variable definition statement>s and within <statements> is replaced by the corresponding newly created <variable name>.

For each <variable definition statement>, a <variable definition> is formed from the <variable definition statement> by omitting the initializing <expression> (if present) and inserted as a <variable definition statement> in place of the original <variable definition statement>. If an initializing <expression> is present, an <assignment statement> is constructed for each <variable name> mentioned in the <local variables of sort> in the order of their occurrence, where <variable name> is given the result of <expression>. These <assignment statement> items are inserted at the front of <statement> items in the order of their occurrence.

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

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

Mapping to abstract syntax

| <compound statement>(*cn, vars, allstats = stats* $\frown$ < *laststatement* >)
=>
mk-*Compound-node*(*Mapping*(*cn*), *Mapping*(*vars*).*toSet*, *undefined*, *empty*,
if *laststatement* $\in$ <terminating statement>

then mk-*Transition*(*Mapping*(*stats*), *Mapping*(*laststatement*))
else mk-*Transition*(*Mapping*(*allstats*), **mk-***Break-node*(*Mapping*(*cn*)))
endif,
empty)

| <compound statement>(*cn*, *vars*, *empty*)
=>
mk-*Compound-node*(*Mapping*(*cn*), *Mapping*(*vars*).*toSet*, *undefined*, *empty*,
mk-*Transition*(*Mapping*(*empty*), **mk-***Break-node*(*Mapping*(*cn*))), *empty*)

F2.2.8.14.2 Transition actions and terminators as statements

Concrete syntax

<assignment statement> :: <assignment>

<return statement> :: <return body>

<call statement> :: <procedure call body> | <remote procedure call body>

Conditions on concrete syntax

∀*rs*∈<return statement>:
(*parentAS0ofKind*(*rs*, <internal procedure definition>)≠*undefined*)∨
(*parentAS0ofKind*(*rs*,<operation definition>)≠*undefined*)

A <return statement> is only allowed within an <internal procedure definition> or within an <operation definition>.

Transformations

The following statements are handled by the Mapping.

<assignment statement> is transformed into the <task>

task <assignment>;

A <call statement> is derived syntax for <procedure call> and is transformed into a <procedure call> with the same <procedure call body>:

call <procedure call body>;

The transform of an <algorithm action statement> and a <return statement> is obtained by dropping the trailing <end>.

Mapping to abstract syntax

| <assignment statement>(*a*) => *Mapping*(*a*)

| <return statement>(*r*) => *Mapping*(*r*)

| <call statement>(*c*) => *Mapping*(*c*)

F2.2.8.14.3 Expressions as statements

Concrete syntax

<expression statement> :: <operator application>

Transformations

let *nn=newName* **in**
<expression statement>(*expr*) =3=>
    <compound statement>(
        < <variable definition statement>
            (< <local variables of sort>(< *nn* >, *expr.staticSort0*, *undefined*) >) >,
        < <assignment statement>(<assignment>(<identifier>(*undefined*, *nn*), *expr*)) >
)

A new <variable name> is created. A <variable definition> is constructed that declares the newly created <variable name> to be of the same sort as the result of <operation application>. Finally, the expression statement is transformed to a <compound statement> consisting of the newly constructed <variable definition>, followed by an <assignment> between the variable with <variable name> and the <operation application>.

F2.2.8.14.4 If statements

Concrete syntax

<if statement> ::
        [<connector name>] <Boolean<expression> <consequence statement> [<alternative statement>]

<consequence statement> = [<non terminating statement>]

<alternative statement> = <non terminating statement> | <terminating statement>

<loop if statement> ::
        [<connector name>]
        <Boolean<expression> <loop consequence statement> [<loop alternative statement>]

<loop consequence statement> = [<statement in loop> | <loop terminating statement>]

Transformations

<if statement>(*connectorname*, *expr*, *cons*, *alt*) =3=>
    <decision statement>(*connectorname*, *expr*,
        <decision statement body>(<
            <algorithm answer part>("true", *cons*),
            if *alt* ≠ *undefined* **then** <algorithm else part>(*alt*) **endif**
            >, *undefined*))

<loop if statement>(*connectorname*, *expr*, *cons*, *alt*) =3=>
    <loop decision statement>(*connectorname*, *expr*,
        <decision statement body>(<
            <algorithm answer part>("true", *cons*),
            if *alt* ≠ *undefined* **then** <algorithm else part>(*alt*) **endif**
            >, *undefined*))

The <if statement> (or <loop if statement>) is transformed to the following <decision statement> (or <loop decision statement> respectively):

decision *boolean expression* {
    (true): *consequence statement*
    else : *alternative statement*
};

where *boolean_expression*, *consequence_statement* and *alternative_statement* represent actual text for the <Boolean expression>, <consequence statement> (<loop consequence statement> respectively) and <alternative statement> (<loop alternative statement> respectively).

F2.2.8.14.5 Decision statements

Concrete syntax

<decision statement> ::
[<connector name>] <question> <decision statement body>
| <if statement>

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

<algorithm answer part> :: <answer> [<non terminating statement> | <terminating statement>]

<algorithm else part> :: <alternative statement>

<loop decision statement> ::
[<connector name>] <question> <loop decision statement body>
| <loop if statement>

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

<loop answer part> ::<answer> { <statement in loop> | <loop terminating statement> }

<loop else part> :: <loop alternative statement>

<loop alternative statement> = <statement in loop> | <loop terminating statement>

A <loop decision statement> differs from a <decision statement> only because it uses <loop decision statement body> instead of <decision statement body> (or <loop if statement> instead of <if statement>) so that <loop break statement> and <loop continue statement> are allowed in the loop. Similarly <loop answer part>, <loop else part>, <loop alternative statement>, <statement in loop> and <loop terminating statement> are used instead of <algorithm answer part>, <algorithm else part>, <alternative statement>, <non terminating statement> and <loop terminating statement>, respectively.

Mapping to abstract syntax

| <decision statement>(*connectorname*, *ques*, *body*)
=> *Mapping*(<loop decision statement>(*connectorname*, *ques*, *body*))

The main difference between a <decision statement> and a <loop decision statement> is the inclusion of <loop continue statement> and <loop break statement>, but these are syntactically excluded from <decision statement>, therefore the *Mapping* for <loop decision statement> is used here.

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

An <algorithm else part> of a <decision statement> represents the *Else-answer* of the *Decision-node* of the *Transition* of the *Compound-node*. If the <alternative statement> is a <terminating statement>, the *Transition* of the *Else-answer* is the *Terminator* represented by the <terminating statement>. If there is no <alternative statement> or no <algorithm else part> in the <decision statement> the *Transition* of the *Else-answer* contains only a *Break-node* with the *Connector-name* of the *Compound-node*. If the <alternative statement> is a <non terminating statement>, the *Transition* of the *Else-answer* is the *Graph-node* represented by the <non terminating statement>.

The mapping of <algorithm answer part> and <algorithm else part> is included in the mapping of <decision statement>. The <if statement> alternative of <decision statement> is removed by a transformation in F2.2.8.14.4 If statements.

```
| <loop decision statement>( connectorname, ques, <loop decision statement body>(answers, elsePart))
    => let cn= if connectorname = undefined then newName else connectorname endif in
        mk-Compound-node(Mapping(cn), ∅, empty,
            mk-Transition(
                mk-Decision-node(
                    mk-Decision-body(
                        Mapping(ques),
                        {   mk-Decision-answer(
                            Mapping(ans),
                            if stat(cc) ∈ <loop continue statement>
                            then mk-Transition(empty,
                                if cc = undefined
                                then mk-Continue-node(cn)
                                else mk-Continue-node(cc)
                                endif)
                            elseif stat ∈ <loop terminating statement>
                            then mk-Transition(empty, Mapping(stat))
                            elseif stat = undefined
                            then mk-Transition(empty,  mk-Break-node(cn))
                            else mk-Transition(Mapping(stat),  mk-Break-node(cn))
                            endif)
                            | aap(ans,stat) in answers },
                        mk-Else-answer(
                            if elsePart ∈ <loop terminating statement>
                            then mk-Transition(empty, Mapping(elsePart))
                            elseif elsePart = undefined
                            then mk-Transition(empty,  mk-Break-node(cn))
                            else mk-Transition(Mapping(elsePart),  mk-Break-node(cn))
                            endif
                        )
                    )
                )
            )
        )
        endlet
```

A <loop decision statement> represents a *Compound-node* in the same way as <decision statement> except that a *Transition* of a *Decision-answer* or *Else-answer* has a *Break-node* for a <loop break statement> or a *Continue-node* for a <loop continue statement>. The mapping of <loop answer part> and <loop else part> is included in the mapping of <loop decision statement>. The <loop if statement> alternative of <loop decision statement> is removed by a transformation in in F2.2.8.14.4 If statements.

F2.2.8.14.6 Loop statements

Concrete syntax

```
<loop statement> ::
        [ <connector name> ] <loop clause>* <loop body statement> [<finalization statement>]

<loop body statement> = <statement in loop>

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

<loop compound statement> ::
        [ <connector name> ] <variable definitions> <loop statements>
```

A <loop compound statement> differs from a <compound statement> because it uses <loop statements> instead of <statements>.

```
<loop statements> ::
```

<statement in loop>+ [<loop terminating statement>]
| <loop terminating statement>

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

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

A <loop terminating statement> allows <loop break statement> and <loop continue statement> as well as <terminating statement>.

<finalization statement> = <statement> | <compound statement>

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

<loop step> = [<expression> | <procedure call body>]

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

<loop variable indication identifier> :: <variable<identifier> [<expression>]

<loop variable definition> :: <aggregation kind> <variable<name> <sort> <expression>

<loop break statement> :: ()

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

Conditions on concrete syntax

∀*lc*∈<loop clause>:
if = *lc*.**s-**<loop step>∈<expression> ∪ <procedure call body> **then**
then *lc*.**s-**<loop variable indication> ≠ *undefined*
end**if**

If a <loop step> of a <loop clause> has an <expression> or <procedure call body> the <loop variable indication> of the <loop clause> shall not be omitted.

∀*ls*∈<loop step>:
(**let** *pd* = *ls*.**s-**<procedure call body>.*calledProcedure0* **in**
pd ≠*undefined*∧*pd*.**s-**<procedure heading>.**s-**<procedure result> = *undefined*
endlet)

The <procedure identifier> in the <procedure call body> of a <loop step> must not refer to a value returning procedure call.

Transformations

l=<loop statement> ∧ *l*. **s-**<connector name> = *undefined* =3=>
l. **s-**<connector name> = *newName*

Generate a name for every unlabelled <loop statement>.

Mapping to abstract syntax

| *l*=<loop statement>(*cn*, *lc*, *body*, *final*)
=> **mk-***Compound-node*(*Mapping*(*cn*),
// *Variable-definition-set* of the *Compound-node*
{ **mk-***Variable-definition*(*Mapping*(*c*.**s-**<loop variable indication>.**s-**<name>),
Mapping(*c*.**s-**<loop variable indication>.**s-**<sort>), **PART**, *undefined*) |
c ∈ *lc*.*toSet*: *c*.**s-**<loop variable indication> ∈ <loop variable definition> },
// *Init-graph-node* list of the *Compound-node*
< **let** *lvi*=*c*.**s-**<loop variable indication> **in**

```
            if lvi ∈ <loop variable definition>
            then mk-Graph-node(mk-Task-node(
                    mk-Assignment(Mapping(lvi.s-<name>), Mapping(lvi.s-<expression>))
                ))
            elseif // lvi ∈ <loop variable definition indicator>
                lvi.s-<expression> ≠ undefined
            then mk-Graph-node(mk-Task-node(
                    mk-Assignment(Mapping(lvi.s-<identifier>), Mapping(lvi.s-<expression>))
                ))
            endif
            endlet
        | c in lc: c.s-<loop variable indication> ≠ undefined  >,
        // While-graph-node of the Compound-node
        mk-While-graph-node(
            <   mk-Expression(c.s-<expression>) | c in lc: c.s-<expression> ≠ undefined >,
            // Finalization-node of the Compound-node
            if final ≠ undefined then mk-Graph-node(Mapping(final)) endif
        ),
        // Transition of the Compound-node
        if body(cc) ∈ <loop continue statement>
        then mk-Transition(empty,
            if cc = undefined
            then mk-Continue-node(Mapping(cn))
            else mk-Continue-node(Mapping(cc))
            endif)
        elseif body ∈ <loop terminating statement>
        then mk-Transition(empty, Mapping(body))
        elseif body = undefined
        then mk-Transition(empty, mk-Continue-node(Mapping(cn)))
        else mk-Transition(< Mapping(body) >, mk-Continue-node(Mapping(cn)))
        endif,
        // Step-graph-node of the Compound-node
        <   let  lvi=c.s-<loop variable indication> in
            let  ls=c.s-<loop step> in
            if lvi ∈ <loop variable definition>
            then mk-Graph-node(mk-Task-node(
                    mk-Assignment (Mapping(lvi.s-<name>), Mapping(ls))
                ))
            else mk-Graph-node(mk-Task-node(
                    mk-Assignment (Mapping(lvi.s-<identifier>), Mapping(ls))
                ))
            endif
            endlet
            endlet
        | c in lc: c.s-<loop step> ≠ undefined >
    ) // end of Compound-node for <loop statement>
```

NOTE – In the above mapping, for the *Init-graph-node* and *Step-graph-node* generated for local variables of the loop, **mk**-*Assignment-node* has the *Mapping* of the <name> of the local variable as the first parameter. This parameter should be a *Variable-identifier*. For a valid derivation of a *Variable-identifier* from the local variable <name>, *Path-item* needs to be extended to include *Connector-name* of a *Compound-node*. Further study is needed.

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

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

in a <loop variable definition> also represents the *Variable-identifier* in any *Step-graph-node* for the <loop step> following the <loop variable indication>.

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

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

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

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

Each <expression> or <procedure call body> of a <loop step> represents an *Expression* of an *Assignment* in a *Task-node* of the *Step-graph-node* list of the *Compound-node*. The *Variable-identifier* of the *Assignment* is the one represented by the <variable identifier> or <variable name> of the preceding <loop variable indication>.

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

The mapping of <finalization statement>, <loop clause>, <loop step>, <loop variable indication>, <loop variable indication identifier> and <loop variable definition> is included in the mapping of <loop statement>.

```
| <loop compound statement>( cn, vars, allstats = stats ⁀ < laststatement >)
    =>
        mk-Compound-node(Mapping(cn), Mapping(vars).toSet, undefined, empty,
            if laststatement ∈ <loop terminating statement>
            then mk-Transition(Mapping(stats), Mapping(laststatement))
            else mk-Transition(Mapping(allstats), mk-Break-node(Mapping(cn)))
            endif,
            empty)
```

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

```
| lbs = <loop break statement>
    => mk-Break-node(Mapping(parentAS0ofKind(lbs, <loop statement>).s-<connector name>))
```

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

```
| lcs = <loop continue statement>(cn)
    => if cn = undefined
        then mk-Continue-node(Mapping(parentAS0ofKind(lcs, <loop statement>).s-<connector name>))
        else mk-Continue-node(Mapping(cn))
        endif,
```

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

F2.2.8.14.7 Break statement

Conditions on abstract syntax

∀*bs*∈*Break-node*: ∃*ls*∈ (*Compound-node*) :
isAncestorAS1(*ls*, *bs*) ∧ (*bs*.**s**-*Connector-name* = *ls*.**s**-*Connector-name*)

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

Concrete syntax

<break statement> :: <connector<name>

Mapping to abstract syntax

| <break statement>(*name*) => **mk**-*Break-node*(*Mapping*(*name*))

F2.2.8.15 Timer

Abstract syntax

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

Conditions on abstract syntax

∀*n* ∈ *Set-node* ∪ *Reset-node*: ∀*d* ∈ *Timer-definition*:
(*d*=*getEntityDefinition1*(*n*.**s**-*Timer-identifier*,**timer**)) ⇒
isActualAndFormalParameterMatched1(*n*.**s**-*Expression*-**seq**, *d*.*formalParameterSortList1*)

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

Concrete syntax

<timer definition> :: <timer definition item>+

<timer definition item> ::
<timer<name> [<sort list>] [<timer default initialization>]

<timer default initialization> :: <Duration<constant expression>

<set> :: <set body>

<set body> :: <set clause>+

<set clause> :: [<Time<expression>] <timer<identifier> <expression>*

<reset> :: <reset body>

<reset body> :: <reset clause>+

<reset clause> :: <timer<identifier> <expression>*

Transformations

<set clause>(*undefined*, *id*, *exprList*)
=8=> <set clause>
(<operator application>("+", **now**, *id.refersto0*.s-<timer default initialization>), *id*, *exprList*)

A <set clause> with no <Time expression> is derived syntax for a <set clause> where <Time expression> is

now + <Duration constant expression>

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

< <set>(< *s* > ⌒ *rest*) > **provided** *rest* ≠ *empty* =1=>
< <set>(< *s* >), <set>(*rest*) >

< <reset>(< *r* > ⌒ *rest*) > **provided** *rest* ≠ *empty* =1=>
< <reset>(< *r* >), <reset>(*rest*) >

A <reset> or a <set> may contain several <reset clause>s or <set clause>s respectively. This is derived syntax for specifying a sequence of <reset>s or <set>s, one for each <reset clause> or <set clause> such that the original order in which they were specified in <reset> or <set> is retained. This shorthand is expanded before shorthand items in the contained expressions are expanded.

< <timer definition>(< *t* > ⌒ *rest*) > **provided** *rest* ≠ *empty* =1=>
< <timer definition>(< *s* >), <timer definition>(*rest*) >

A <timer definition> may contain several <timer definition item>s. This is derived syntax for specifying a sequence of <timer definitions>s, one for each <timer definition item>. This is an auxiliary transformation.

Mapping to abstract syntax

| <timer definition>(< *item* >) => *Mapping*(*item*)

| <timer definition item>(*name*, *sortList*, *) =>
mk-*Timer-definition*(*Mapping*(*name*), *Mapping*(*sortList*))

| <set>(< *clause* > => **mk-***Graph-node*(*Mapping*(*clause*))

| <set clause>(*expr*, *id*, *params*) => **mk-***Set-node*(*Mapping*(*expr*), *Mapping*(*id*), *Mapping*(*params*))

| <reset>(< *clause* > => **mk-***Graph-node*(*Mapping*(*clause*))

| <reset clause>(*id*, *params*) => **mk-***Reset-node*(*Mapping*(*id*), *Mapping*(*params*))

F2.2.9 Data

F2.2.9.1 Data definitions

Concrete syntax

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

<sort> =
<basic sort> [<range condition>]
| <anchored sort>
| <pid sort>
| <inline data type definition>
| <inline syntype definition

<basic sort> =
        <datatype<type expression>
    | <as signal>
    | <as interface>
    | <as channel>
    | <as gate>
    | <syntype>

<anchored sort> :: **this** | **parent** | <this<basic sort>|<parent<basic sort>

<pid sort> = <sort<identifier>

<inline data type definition> :: [<data type specialization>] <data type definition body>

<inline syntype definition> :: <basic sort> { <default initialization> [<constraint>] | <constraint> }

Conditions on concrete syntax

∀*s*∈<sort>: ∀*bs* = *s*.**s-**<basic sort>:∀*rc* = *s*.**s-**<range condition>:(*rc* ≠ *undefined*) ⇒ *bs*∈<literal list>

A <range condition> after a <basic sort> of a sort is only valid if the <basic sort> has been constructed using the literal data type constructor and the **constants (**<range condition>**)** is valid as a <constraint> for the <basic sort>, or if the **size (**<range condition>**)** is valid as a <constraint> for the <basic sort>.

Transformations

anchSort = <anchored sort>(**this**)
=>
<basic sort>(<datatype(*parentAS0ofKind*(*anchSort*, <data type definition>))

anchSort = <anchored sort>(**parent**)
=>
parentAS0ofKind(*anchSort*, <data type definition>).*specialization0*.**s-**<type expression>.*BaseType0*

An <anchored sort> without a <basic sort> is a shorthand for specifying a <basic sort> with the name of the data type definition (or syntype definition) in the context where the <anchored sort> occurs.

let *nn* = *newName* **in**
    s = <sort>(*bsort*: <basic sort>, *rc*: <range condition>)!
    =>
    <syntype <identifier>(*empty*, *nn*)>
and // add the implicit syntype definition
    let *parent* = *s.surroundingScopeUnit0* **in**
        entities=parent.getEntities
        =>
        entities ⌒ <syntype definition>(*empty*,
        <syntype definition syntype>(*nn*, *bsort*, *undefined*, *rc*))
    endlet
endlet

A <sort> that is a <basic sort> with a <range condition> is derived concrete syntax for a <syntype> of an implied <syntype definition> having an anonymous name for the <syntype name> and the <basic sort> as the <parent sort identifier>. This anonymously named <syntype definition> is defined with no <default initialization> and with its elements restricted by a <constraint> based on the <range condition>.

NOTE – The conditions on concrete syntax described below (in section F2.2.9.8.4Constraint) apply to the <syntype> and its <constraint> based respectively on <basic sort> and <range condition>.

let *nn* = *newName* **in**
s = <inline data type definition>(*dtspecial*: <data type specialization>, *dtdbody*: <data type definition body>)
=>
<basic sort>(<datatype<type expression>(<identifier>(*empty*,*nn*)))>
and // add the implicit data type definition
    let *parent* = *s.surroundingScopeUnit0* **in**
        entities=parent.getEntities
        =>

entities ⌒ <data type definition>(*empty*, *undefined*, <data type *nn*>, *dtspecial*, *dtdbody*, *undefined*)

endlet

endlet

An <inline data type definition> is derived concrete syntax for a <basic sort> of an implied <data type definition> having an anonymous name. This anonymously named <data type definition> is derived from the <inline data type definition> by inserting **type** and the anonymous name after **value** in the <inline data type definition>. Each <inline data type definition> defines a different implied <data type definition>.

let *nn* = *newName* **in**
s = <inline syntype definition>(*bsort*: <basic sort>, *init*: <default initialization>, *constr*: <constraint>)}
=> <syntype <identifier>(*empty*, *nn*)>
and // add the implicit syntype definition
let *parent* = *s.surroundingScopeUnit0* **in**
entities=parent.getEntities
=>

entities ⌒ <syntype definition>(*empty*, <syntype definition syntype>(*nn*, *bsort*, *init*, *constr*))

endlet
endlet

An <inline syntype definition> is derived concrete syntax for a <basic sort> of an implied <syntype definition> having an anonymous name. This anonymously named <syntype definition> is derived from the <inline syntype definition> by inserting the anonymous name and <equals sign> after syntype in the <inline syntype definition>.

F2.2.9.2 Data type definition

Abstract syntax

Data-type-definition	=	*Value-data-type-definition*
	|	*Interface-definition*
Value-data-type-definition	::	*Sort* [*Data-type-identifier*] *Literal-signature*-**set** [*Null-literal-signature*] *Static-operation-signature*-**set** *Dynamic-operation-signature*-**set** *Procedure-definition*-**set** *Data-type-definition*-**set** *Syntype-definition*-**set** [*Default-initialization*] [*Abstract*]

Concrete syntax

<data type definition> ::
<package use clause>* <type preamble> <data type heading> [<data type specialization>]
[<data type definition body>]

<data type heading> ::
<data type><name> [<formal context parameters>] [<virtuality constraint>]

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

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

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

<operation definitions> :: <operation definition item>*

Conditions on concrete syntax

∀*dtd*∈<data type definition>:∀*fcp*∈ <formal context parameter>:
fcp∈*dtd.localFormalContextParameterList0.toSet* ⇒
fcp∈<sort context parameter> ∪<synonym context parameter list>

A <formal context parameter> of <formal context parameters> must be either a <sort context parameter> or a <synonym context parameter list>.

∀*anchSort*∈<anchored sort>:
parentAS0ofKind
(*anchSort*, <data type definition>) ≠ *undefined* ∧ (*anchSort*.**s-**<basic sort> ≠ *undefined*⇒
parentAS0ofKind(*anchSort*, <data type definition>) =
getEntityDefinition0(*anchSort*.**s-**<basic sort>, **sort**))

An <anchored sort> is legal concrete syntax only if it occurs within a <data type definition>. The <basic sort> in the <anchored sort> must name the <sort> introduced by the <data type definition>.

∀*sid*∈<pid sort>:*isPidSort0*(*sid*)

The <sort identifier> in a <pid sort> must reference a pid sort.

Transformations

s = <system type definition>(*uses*, <system type heading>(*undefined*, *name*, *heading*)
=1=>
<system type definition>(*uses*, <system type heading>(*s.fullQualifier0*, *name*, *heading*))

<*s* = <system type definition>(*uses*, <system type heading>(*qual*, *name*, *heading*)) >
=9=>
< *s* > ⌒
< <interface definition>(*undefined*,<interface heading>(*name*, *empty*, *undefined*),
<interface specialization>(
< <type expression>
(<identifier>(<path item>(**system type**, *name*), *name*)) >), *empty*, *undefined*) >

< *s* = <block type definition>(*uses*, <block type heading>(*qual*, *name*, *heading*)) >
=9=>
< *s* > ⌒
< <interface definition>(*undefined*,<interface heading>(*name*, *empty*, *undefined*),
<interface specialization>(
< <type expression>
(<identifier>(<path item>(**block type**, *name*), *name*)) >), *empty*, *undefined*) >

< *s* = <process type definition>(*uses*, <process type heading>(*qual*, *name*, *heading*)) >
=9=>
< *s* > ⌒
< <interface definition>(*undefined*,<interface heading>(*name*, *empty*, *undefined*),
<interface specialization>(
< <type expression>
(<identifier>(<path item>(**process type**, *name*), *name*)) >), *empty*, *undefined*) >

< *s* = <textual typebased system definition>
(<typebased system heading>(name, <type expression>(base, *))) >
=9=>
< *s* > ⌒
< <interface definition>(*undefined*,<interface heading>(*name*, *empty*, *undefined*),
<interface specialization>(< <type expression>(*base*) >), *empty*, *undefined*) >

< *s* = <textual typebased block definition>
(<typebased block heading>(name, <type expression>(base, *))) >
=9=>

< *s* > ⌒
< <interface definition>(*undefined*,<interface heading>(*name*, *empty*, *undefined*),
<interface specialization>(< <type expression>(*base*) >), *empty*, *undefined*) >

< *s* = <textual typebased process definition>(name, <type expression>(base, *)) >
=9=>
< *s* > ⌒
< <interface definition>(*undefined*,<interface heading>(*name*, *empty*, *undefined*),
<interface specialization>(< <type expression>(*base*) >), *empty*, *undefined*) >

a = <agent structure>(*iset*, *entities*, *interaction*)=>
<agent structure>(*iset*, *entities* ⌒
< <interface definition>(*undefined*,<interface heading>(*name*, *empty*, *undefined*),
<interface specialization>(*a.gateDefinitionSet0.impliedBases*, *empty*,
a.gateDefinitionSet0.gatesignalset0)) >, *interaction*)

Interfaces are implicitly defined by the agent, its state machine and agent type definitions. The implicitly defined interface has the same name as the agent or agent type that defined it.

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

The interface defined by a state machine contains in its <interface specialization> all interfaces given in the incoming signal list associated with explicit or implicit gates of the state machine. The interface also contains in its <interface use list> all signals, remote variables and remote procedures given in the incoming signal list associated with explicit or implicit gates of the state machine.

The interface defined by an agent or agent type contains in its <interface specialization> the interface defined by the composite state representing its state machine.

The interface defined by a type based agent or service contains in its <interface specialization> the interface defined by its type.

NOTE – To avoid cumbersome text, the convention is used that the phrase "the pid sort of the agent A" is often used instead of "the pid sort defined by the interface implicitly defined by the agent A" when no confusion is likely to arise.

Mapping to abstract syntax

| <data type definition>(*, *, <data type heading>(*name*, *, *), *base*, *body*) =>
mk-*Value-data-type-definition*(*Mapping*(*name*), *Mapping* (*base*),
{ *e* ∈ *Mapping*(*entities*).*toSet*: *e* ∈ *Literal-signature* },
{ *e* ∈ *Mapping*(*entities*).*toSet*: *e* ∈ *Static-operation-signature* },
{ *e* ∈ *Mapping*(*entities*).*toSet*: *e* ∈ *Data-type-definition* },
{ *e* ∈ *Mapping*(*entities*).*toSet*: *e* ∈ *Syntype-definition*})

| <data type definition body>(*entities*, *ctor*, *operations*, *) => *Mapping*(*entities*) ⌒ *Mapping*(*ctor*)

| <operations>(*, *bodies*) => *Mapping*(*bodies*)

Auxiliary functions

The function *localDataTypeDefinitionSet0* gets the local defined <data type definition>s in a scope unit.

localDataTypeDefinitionSet0(*su*: *SCOPEUNIT0*): <data type definition>-**set**=$_{\text{def}}$
{*d*∈<data type definition>: *d.surroundingScopeUnit0* = *su*}

The function *impliedBases* computes the implied base interface.

impliedBases(*g*: <gate in definition>* ∪ <gate constraint> ∪ <signal list item>):
<type expression>+ =$_{def}$
case *g* **of**
| < <textual gate definition>(*, *c1*, *c2*) > ⌒ *tail* =>
impliedBases(*c1*) ⌒ *impliedBases*(*c2*) ⌒ *impliedBases* (*tail*)
| < <textual interface gate definition> (*, *ident*) > ⌒ *tail* => < *ident* > ⌒ *impliedBases* (*tail*)
| <gate constraint>(**in**, *, signals) => *bigSeq*(<*impliedBases*(*s*) | *s* **in** *signals*>)
| <signal list item>(**interface**, *ident*) => < *ident* >
otherwise
empty
endcase

The function *gatesignalset0* computes the **in** signals of a gate list.

gatesignalset0(*g*: <gate in definition>* ∪ <gate constraint> ∪ <signal list item>): *SIGNAL0*=$_{def}$
case *g* **of**
| < <textual gate definition>(*, *c1*, *c2*) > ⌒ *tail* =>
gatesignalset0(*c1*) ∪ *gatesignalset0*(*c2*) ∪ *gatesignalset0* (*tail*)
| < <textual interface gate definition> (*, *ident*) > ⌒ *tail* =>
let *fd* = *getEntityDefinition0*(*ident*, **interface**) **in**
fd.usedSignalSet0∪{*sd.identifier0*: *sd*∈ *fd.definedSignalSet0*} ∪ *tail.gatesignalset0*
| <gate constraint>(**in**, *, *signals*) => *signals.signalSet0*
otherwise
∅
endcase

F2.2.9.3 Interface type

Abstract syntax

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

Concrete syntax

<interface definition> ::
<package use clause>* [<virtuality>] <interface heading> [<interface specialization>]
<entity in interface>*
| <signal list definition>

<interface heading> ::
<interface<name> [<formal context parameters>] [<virtuality constraint>]

<entity in interface> =
<signal definition list>
| <interface variable definition>
| <interface procedure definition>
| <interface use list>

<interface use list> :: [<signal list>]

<interface variable definition> :: <remote variable<name>+ <sort>

<interface procedure definition> :: <remote procedure<name> <procedure signature>

<as interface> :: <interface<identifier>

Further study is needed for the handling of <as interface>.

Conditions on concrete syntax

∀*fd*∈<interface definition>:
isRestrictedByInterface0(*fd*.**s-**<entity in interface>**s-**<interface use list>.**s-**<signal list item>**-seq**)

The <signal list> in an <interface definition> shall only contain <signal identifier>s, <remote procedure identifier>s, <remote variable identifier>s and <signal list identifiers>. If a <signal list identifier> is part of the <signal list> it must also respect this restriction.

∀*ind*∈<interface definition>:∀*fcp*∈<formal context parameter>:
fcp∈*ind.localFormalContextParameterList0* ⇒
fcp∈<signal context parameter list> ∪<remote procedure context parameter> ∪
<remotevariable context parameter list> ∪ <sort context parameter>

The <formal context parameters> shall only contain <signal context parameter list>, <remote procedure context parameter>, <remotevariable context parameter list> or <sort context parameter>.

Mapping to abstract syntax

| <interface definition>(*, *, <interface heading>(*name*, *, *), *spec*, *entities*, *)
=> **mk-***Interface-definition*(*Mapping*(*name*), *Mapping*(*spec*)
{ *e* ∈ *Mapping*(*entities*).*toSet*: (*e* ∈ *Signal-definition*)})

Auxiliary functions

The function *localInterfaceDefinitionSet0* gets the local defined interface definition list of a scope unit.

localInterfaceDefinitionSet0(*su*: *SCOPEUNIT0*): <interface definition>-**set**=$_{def}$
{*d*∈<interface definition>:*d.surroundingScopeUnit0 = su*}

The function *isRestrictedByInterface0* decides if a <signal list> only contains <signal identifier>s, <remote procedure identifier>s, <remote variable identifier>s and <signal list identifiers>. If a <signal list identifier> is part of the <signal list> it must also respect this restriction.

isRestrictedByInterface0(*sl*: <signal list item>*): *BOOLEAN* =$_{def}$
if *sl* = *empty* **then** *true*
else
case *sl*.*head*.**s-implicit of**
| {**signal**, **remote procedure**, **remote variable**}=>*isRestrictedByInterface0*(*sl.tail*)
| {**signallist**}=>
let *sl'=getEntityDefinition0*(*sl.head*.**s-**<identifier>, **signallist**).**s-**<signal list item>**-seq in**
isRestrictedByInterface0(*sl'*)∧*isRestrictedByInterface0*(*sl.tail*)
otherwise=>*false*
endcase
endif

F2.2.9.4 Specialization of data types

Concrete syntax

<data type specialization> :: <data type<type expression>+ <renaming>

<renaming> :: <rename list>

<rename list> = <rename pair>*

<rename pair> =
<rename pair operation name> | <rename pair literal name>

<rename pair operation name> :: <operation name> <base type<operation name>

<rename pair literal name> :: <literal name> <base type<literal name>

<interface specialization> :: <interface<type expression>+

Conditions on concrete syntax

∀*dataDef*∈<data type definition>:∀*superTypeDef*∈<data type definition>:
isSubtype0(*dataDef*, *superTypeDef*) ⇒
superTypeDef.**s-**<data type definition body>.**s-**<data type constructor>= *undefined* ∨
isSameConstructorKind0(
superTypeDef.**s-**<data type definition body>.**s-**<data type constructor>,
dataDef.**s-**<data type definition body>.**s-**<data type constructor>)

The <data type constructor> must be of the same kind as the <data type constructor> used in the <data type definition> of the sort referenced by <data type type expression> in the <data type specialization>.

∀*rn*∈<renaming>:
let *lnl* = << *rp*.**s-**<literal name>, *rp*.**s2-**<literal name>> | *rp* **in** *rn*.**s-**<rename pair>**-seq** > **in**
let *onl* = << *rp*.**s-**<operation name>, *rp*.**s2-**<operation name>> | *rp* **in** *rn*.**s-**<rename pair>**-seq** > **in**
(∀*i*,*j*∈1..*lnl.length*: *i*≠*j*⇒ *lnl*[*i*]≠ *lnl*[*j*])∧
(∀*i*,*j*∈1..*onl.length*: *i*≠*j*⇒ *onl*[*i*]≠ *onl*[*j*])

All <literal name>s and all <base type literal name>s in a <rename list> must be distinct. All <operation name>s and all <base type operation name>s in a <rename list> must be distinct.

∀*sp*∈<data type specialization>:
(**let** *bt* = *sp*. **s-**<type expression>.*BaseType0* **in**
let *onl* = <*rp*.**s2-**<operation name> |
rp **in** *sp*. **s-**<renaming>.**s-**<rename pair>**-seq**: *rp*∈<rename pair operation name>> **in**
∀*on*∈<operation name>: *on* **in** *onl*⇒
(∃*os*∈<operation signature>: *os.surroundingScopeUnit0* = *bt* ∧ *on* = *os.name0*)
endlet)

A <base type operation name> specified in a <rename list> must be an operation with <operation name> defined in the data type definition defining the <base type> of <data type type expression>.

∀*sp*∈<data type specialization>:∀*te*∈ *sp*.**s-**<type expression>:

¬*isAbstractType0*(*te*.*BaseType0*) ⇒∀*te'*∈ *sp*.**s-**<type expression>\{*te*}:*isAbstractType0*(*te'*.*BaseType0*)

At most one of the <data type<type expression>s is allowed to be not abstract.

Transformations

<data type definition>(*use*, *preamble*, *heading*, *spec*, *undefined*)
=>
<data type definition>(*use*, *preamble*, *heading*, *spec*,
<data type definition body>(*undefined*, *undefined*, *inheritedOperations*(*spec*), *undefined*))

<data type definition>
(*use*, *preamble*, *heading*, *spec*, <data type definition body>
(*entities*, *constr*, <operations>
(<operation signatures> (<operator list>(*ops*), <method list> (*meths*)), *refs*, *defs*), *init*))
=>
<data type definition>(*use*, *preamble*, *heading*, spec,
<data type definition body>(*undefined*, *undefined*,
<operations>(<operation signatures>(
<operator list>(*ops*) ⌒
inheritedOperations(*spec*).s-<operation signatures>.s-<operator list>,
<method list> (*meths*) ⌒
inheritedOperations(*spec*).s-<operation signatures>.s-<method list>),
refs, *defs*), *undefined*))

The model for specialization in clause 8.4 of [ITU-T Z.102] is used, augmented as follows.

A specialized data type is based on another (base) data type by using a <data type definition> in combination with a <data type specialization>. The sort defined by the specialization is disjoint from the sort defined by the base type.

If the sort defined by the base type has literals defined, the literal names are inherited as names for literals of the sort defined by the specialized type unless literal renaming has taken place for that literal. Literal renaming has taken place for a literal if the base type literal name appears as the second name in a <rename pair> in which case the literal is renamed to the first name in that pair.

If the base type has operators or methods defined, the operation names are inherited as names for operators or methods of the sort being defined, subject to the restrictions stated in clause 8.4.1 of [ITU-T Z.102], unless the operator or method has been declared as private (see clause 12.1.8.4 of [ITU-T Z.104]) or operation renaming has taken place for that operator or method. Operation renaming has taken place for an operator or method if the inherited operation name appears as the second name in a <rename pair> in which case the operator or method is renamed to the first name in that pair.

When several operators or methods of the <base type> of <sort type expression> have the same name as the <base type operation name> in a <rename pair>, then all of these operators or methods are renamed.

In every occurrence of an <anchored sort> in the specialized type, the <basic sort> is replaced by the subsort.

The argument sorts and result of an inherited operator or method are the same as those of the corresponding operator or method of the base type, except that in every <argument> containing an <anchored sort> in the inherited operator or method the <basic sort> is replaced by the subsort.

Mapping to abstract syntax

| <data type specialization>(*base*, *) => *Mapping*(*base*)

| <interface specialization>(*bases*, *) => *Mapping*(*bases*)

Auxiliary functions

The function *isRenamedBy0* determine if a <literal signature> or an <operation signature> is renamed by a <specialization>.

isRenamedBySpec0(*sn*:<literal signature>∪<operation signature>, *spec*:<specialization>):
BOOLEAN=$_{\text{def}}$
(∃*rp*∈<rename pair>:
(*rp*. *parentAS0*.*parentAS0* = *spec*)∧
(*rp*.**s2-**<literal name> = *sn*.*name0* ∨ *rp*.**s2-**<operation name> = *sn*.*name0*))

The function *isSameConstructorKind0* is used to determine if two data type constructor items are of the same kind.

isSameConstructorKind0(*c1*:<data type constructor>, *c2*:<data type constructor>): *BOOLEAN* =$_{\text{def}}$
(*c1*∈<literal list>∧*c2*∈<literal list>)∨
(*c1*∈<structure definition>∧*c2*∈<structure definition>)∨
(*c1*∈<choice definition>∧*c2*∈<choice definition>)

Sort compatibility determines when a sort can be used in place of another sort, and when it cannot. The function *isSortCompatible0* is used to determine if the first sort is sort compatible to the second one.

isSortCompatible0(*sort1*:<sort>, *sort2*:<sort>): *BOOLEAN* =$_{\text{def}}$
isSameSort0(*sort1*, *sort2*) ∨
isDirectlySortCompatible0(*sort1*, *sort2*) ∨

```
    (isPidSort0(sort2) ∧
        ∃sort3∈<sort>: isSortCompatible0(sort1, sort3) ∧ isSortCompatible0(sort3, sort2))
```

The function *isSameSort0* is used to determine if the given two sorts are the same.

```
isSameSort0(sort1: <sort>, sort2: <sort>): BOOLEAN =def
    sort1 = sort2 ∨
    (sort1∈<basic sort>∧ sort2∈<basic sort>∧
    getEntityDefinition0(sort1,sort).derivedDataType0=
        getEntityDefinition0(sort2, sort).derivedDataType0)∨
    (sort1∈<anchored sort> ∧ sort2∈<anchored sort> ∧
    parentAS0ofKind(sort1, <data type definition>)=parentAS0ofKind(sort2,<data type definition>))∨
    (sort1∈<pid sort> ∧ sort2∈<pid sort> ∧
    getEntityDefinition0(sort1, sort) = getEntityDefinition0(sort2, sort))
```

Determine if two sort lists are the same.

```
isSameSortList0(sl, sl': <sort>*): BOOLEAN =def
    (sl.length = sl'.length) ∧
    (∀i∈1.. sl.length: isSameSort0(sl[i], sl'[i]))
```

The function *isDirectlySortCompatible0* is used to determine if the sort in the first argument is directly sort compatible to the one in the second.

```
isDirectlySortCompatible0(sort1:<sort>, sort2:<sort>): BOOLEAN =def
    (sort1∈<anchored sort> ∧ sort1.s-<basic sort> = sort2) ∨
    (sort1 ∈ <pid sort> ∧ isSubSort0(sort1, sort2))
```

The function *isPidSort0* is used to determine if a sort is a pid sort.

```
isPidSort0(sort: <sort>): BOOLEAN =def
    getEntityDefinition0(sort,sort)∈<interface definition>
```

The function *isSubSort0* is used to determine if the sort given in the first argument is a super sort of the one in the second.

```
isSubSort0(sort1: <sort>, sort2: <sort>): BOOLEAN =def
    let td1 = getEntityDefinition0(sort1, sort) in
    let td2 = getEntityDefinition0(sort2, sort) in
        (td1∈<interface definition>⇒isSubtype0(td1, td2)) ∧
        (td1∈<data type definition>∪<syntype definition>⇒
            isSubtype0(td1.derivedDataType0, td2.derivedDataType0))
    endlet
```

The function *inheritedOperations* computes the names of the operations inherited from the base type.

```
inheritedOperations(spec: <specialization>): <operations> =def
    let ops = { o ∈ <operation signature>: isVisibleThroughBaseType0(o, spec.parentAS0)
        ∧ o.parentAS0 ∈ <operator list>} in
    let meths = { o ∈ <operation signature>: isVisibleThroughBaseType0(o, spec.parentAS0)
        ∧ o.parentAS0 ∈ <method list>in
        <operations>(<operation signatures>(<operator list>(doRename (ops, spec)),
                                            <method list> (doRename(meths, spec))),
                    undefined, undefined)
    endlet
```

The function *doRename* adjusts an operation for use in the derived type.

```
doRename(o: <operation signature>, spec: <data type specialization>): <operation signature>=def
    case o of
    | <operation signature>(preamble, name, arguments, result) =>
        if isRenamedBySpec0(o, spec) then
            let name1 =    if isRenamedBySpec0(o, spec) then
```

```
                    take({ n ∈ <name>: isRenamedBy0(n, name)}
                else name endif
        in
            mk-<operation signature>( preamble, name1,
                < specializeAnchoredSort0(a, spec): a in arguments>,
                                specializeAnchoredSort0(result, spec))
        endlet
    otherwise
        undefined
    endif
```

The function *specializeAnchoredSort0* replaces every <anchored sort> with the specialized sort.

```
specializeAnchoredSort0(arg: <argument> ∪ <result> ∪ <sort>, spec: <data type specialization>):
        <argument> ∪ <result> ∪ <sort> =def
    case arg in
    | <argument>( kind, sort) =>
        mk-<argument>( kind, specializeAnchoredSort0(sort, spec))
    | <result>(sort) => mk-<result>(specializeAnchoredSort0(sort, spec))
    | <anchored sort>(*) => mk-<anchored sort>(spec.parentAS0.name0)
    otherwise
        arg
    endcase
```

F2.2.9.5 Operations

Abstract syntax

Static-operation-signature	=	*Operation-signature*
Dynamic-operation-signature	=	*Operation-signature*
Operation-signature	::	*Operation-name Formal-argument** [*Operation-result*] *Procedure-identifier*
Formal-argument	=	*Argument*
Operation-result	=	*Sort-reference-identifier*
Argument	=	*Sort-reference-identifier*

Conditions on abstract syntax

Concrete syntax

<operation signatures> = [<operator list>] [<method list>]

<operator list> :: <operation signature>+

<method list> :: <operation signature>+

<operation signature> ::= <operation preamble> <operation name> [<arguments>] [<result>]

<operation preamble> ::= [<visibility> [<virtuality>]] | [<virtuality> [<visibility>]]

<arguments> :: <argument> +

<argument> = <formal parameter>

<formal parameter> :: <parameter kind> <sort>

<result> :: <sort>

Conditions on concrete syntax

∀*os*∈<operation signature>: *os.EntityKind0* = **operator** ⇒ *os*.**s-**<result> ≠ *undefined*

Transformations

<operation signature>(*pre*, *name*, *undefined*, *result*) =1=> <operation signature>(*pre*, *name*, *empty*, *result*)

o = <operation signatures>((

<operator list>(*operations*),

<method list> (<operation signature>(*pre*, *name*, *args*, *result*) ⌒ *rest*))

=2=>

<operation signatures>(<operator list>(*operations* ⌒

<operation signature>(*pre*, *name*,

<argument>(**inout,** *parentAS0ofKind*(*o*, *SCOPEUNIT0*).*identifier0*) ⌒

args, *result*))), <method list>(*rest*))

If <operation signature> is contained in a <method list> this is derived syntax and is transformed as follows: An <argument> is constructed from the <parameter kind> in/out, and the <sort identifier> of the sort being defined by the enclosing <data type definition>. If there are no <arguments>, then <arguments> is formed from the constructed <argument> and inserted into the <operation signature>. If there are <arguments>, the constructed <argument> is added to the start of the original list of <argument>s in the <arguments>.

<formal parameter>(*undefined*, *sort*) => <formal parameter>(**in**, *sort*)

An <argument> (<formal parameter>) without an explicit <parameter kind> has the implicit <parameter kind> **in**.

Mapping to abstract syntax

| <operation signatures>(*operators*, *) => *Mapping*(*operators*).*toSet*

|*os* = <operation signature>(<operation preamble>(*undefined*, *undefined*), *name*, *arguments*, *result*) =>
mk-*Static-operation-signature*(*Mapping*(*name*), *Mapping*(*arguments*), *Mapping*(*result*),
Mapping(*os.operatorProcedureName*))

Auxiliary functions

The function *operatorProcedureName* associates each operation signature with its implicit anonymous procedure name.

controlled *operatorProcedureName*: <operation signature> → <identifier>
initially ∀ o ∈ <operation signature>: o.*operatorProcedureName* = *empty*

Get the list of the arguments of an operation signature.

operationSignatureParameterList0(*os*: <operation signature>):<argument>* =$_{def}$
(*os*.**s**-<argument>-**seq**)

F2.2.9.6 Data type constructors

Concrete syntax

<data type constructor> = <literal list> | <structure definition> | <choice definition>

F2.2.9.6.1 Literals

Abstract syntax

Literal-signature	::	*Literal-name* *Result* *Literal-natural*
Literal-natural	=	*Nat*

Concrete syntax

<literal list> :: [<visibility>] <literal signature>+

<literal signature> = <literal name> | <named number>

<named number> :: <literal name> <Natural<simple expression>

Conditions on concrete syntax

∀*num1*, *num2*∈<named number>: *num1*. *parentAS0*=*num2*. *parentAS0*∧*num1*≠*num2*⇒
num1.**s-**<simple expression>.*value0*≠ *num2*.**s-**<simple expression>.*value0*

Each result of <Natural simple expression> occurring in a <named number> must be unique among all <literal signature>s in the <literal list>.

Transformations

<literal list>(*head* ⌒ < *name* > ⌒ *tail*)
provided *name* ∈ <literal name>
∧ ∀ *n* ∈ <literal name>: ¬ *n* **in** *head*
=>
<literal list>(*head* ⌒ <named number>(*name*, *nextNumber*(*head*)) ⌒ *tail*)

A <literal name> in a <literal list> is derived syntax for a <named number> containing the <literal name> and containing a <Natural simple expression> denoting the lowest possible non-negative Natural value not occurring in any other <literal signature>s of the <literal list>. The replacement of <literal name>s by the <named number>s takes place one by one from left to right.

b = <data type definition body>(*entities*, s = <literal list>(*, *), <operations>(
<operation signatures>(<operator list>(*operators*), *methods*), *refs*, *defs*), *init*)
provided ¬ *b*.*parentAS0*.*identifier0*.*implicitSignaturesAdded*
=2=>
let *nopreamble* = <operation preamble>(*undefined*, *undefined*) **in**
let *sort* = b.*parentAS0*.*identifier0* **in**
let *arg* = <argument>(**in**, <anchored sort>(*sort*)) **in**
let *resbool* = <result>(<identifier>(<qualifier>(<name>("Predefined")), <name>("Boolean"))) **in**
let *reslit* = <result>(<anchored sort>(*sort*)) **in**
let *newoperators* =
< <operation signature>(*nopreamble*, <name>("<"), < *arg*, *arg* >, *resbool*),
<operation signature>(*nopreamble*, <name>(">"), < *arg*, *arg* >, *resbool*),
<operation signature>(*nopreamble*, <name>(">="), < *arg*, *arg* >, *resbool*)),
<operation signature>(*nopreamble*, <name>("<="), < *arg*, *arg* >, *resbool*),
<operation signature>(*nopreamble*, <name>("first"), < >, *reslit*),
<operation signature>(*nopreamble*, <name>("last"), < >, *reslit*),
<operation signature>(*nopreamble*, <name>("pred"), < *arg* >, *reslit*),
<operation signature>(*nopreamble*,<name>("succ"), < *arg* >, *reslit*),
<operation signature>(*nopreamble*, <name>("num"), < *arg*, *arg* >,
<result>(<identifier>(<qualifier>(<name>("Predefined")), <name>("Natural")))) >
in
<data type definition>(*entities*, *s*, <operations>(<operation signatures>(
<operator list>(*operators* ⌒ *newoperators*), *methods*), *refs*, *defs*), *init*)
endlet // *newoperators*
endlet // *reslit*
endlet // *resbool*
endlet // *arg*
endlet // *sort*
endlet // *nopreamble*
and
b.*parentAS0*.*identifier0*.*implicitSignaturesAdded*:= *true*

A literal list is derived syntax for the definition of operators that establish an ordering of the elements in the sort defined by the <literal list>:

a) operators that compare two data items with respect to the established ordering;

b) operators that return the first, last, next, or previous data item in the ordering; and

c) an operator that gives the position of each data item in the ordering.

A <data type definition> introducing a sort named S by a <literal list> implies a set of Static-operation-signatures equivalent to the explicit definitions in the following <operator list>:

```
"<"     ( this S, this S ) -> Boolean;
">"     ( this S, this S ) -> Boolean;
"<="    ( this S, this S ) -> Boolean;
">="    ( this S, this S ) -> Boolean;
first                      -> this S;
last                       -> this S;
succ    ( this S )         -> this S;
pred    ( this S )         -> this S;
num     ( this S )         -> Natural;
```

where Boolean is the predefined Boolean sort and Natural is the predefined Natural sort.

The <literal signature> items in a <data type definition> are nominated in ascending order of the <Natural simple expression> items. For example,

```
literals C = 3, A, B;
```

implies A<B and B<C.

The comparison operators "<" (">","<=",">=") represent the standard less-than (greater-than, less-or-equal-than, and greater-or-equal-than) comparison between the <Natural simple expression>s of two literals. The operator first returns the first data item in the ordering (the literal with the lowest <Natural simple expression>). The operator last returns the last data item in the ordering (the literal with the highest <Natural simple expression>). The operator pred returns the preceding data item, if one exists, or the last data item, otherwise. The operator succ returns the successor data item in the ordering, if one exists, or the first data item, otherwise. The operator num returns the Natural value corresponding to the <Natural simple expression> of the literal.

This transformation is defined as part of the mapping.

Mapping to abstract syntax

```
| <literal list>(*,signatures) => Mapping(signatures)

| <named number>(name, number) =>
    mk-Literal-signature
        (   mk-Literal-name(Mapping(name)),
            mk-Result(Mapping(
                identifier0(parentAS0ofKind(<named number>, <data type definition>))))
            mk-Literal-natural(Mapping(number)))
```

Auxiliary functions

```
visibility0(s:<operation signature>∪<literal signature>):<visibility>=def
    if s∈<operation signature> then s.s-<visibility>
    else s.parentAS0.s-<visibility>
```

The function *nextNumber* computes the next available number for a literal list.

```
nextNumber(literals: <literal signature>*): <simple expression> =def
    if literals = empty then
        mk-<identifier>(empty, <name>("0"))
    elseif literals.tail.nextNumber ≠ undefined then
        literals.tail.nextNumber
    elseif literals.head ∈ <named number> then
        <operator application>(<operation identifier>(empty, <name>("+")),
            < literals.head.s-<simple expression>,
            mk-<operand5>(undefined, mk-<identifier>(empty, <name>("0")))
```

else
undefined
endif

The function *implicitSignaturesAdded* records whether the implicit signatures for literal lists have been added into a data type.

controlled *implicitSignaturesAdded*: <identifier> → *BOOLEAN*

F2.2.9.6.2 Structure data types

Concrete syntax

<structure definition> :: [<visibility>] <field list>

<field list> = <field>+

<field> =
<optional field> | <mandatory field>

<optional field> :: <fields of sort>

<mandatory field> :: <fields of sort> [<field default initialization>]

<field default initialization> :: <constant expression>

<fields of sort> :: [<visibility>] <fos gen field of kind>+ <field sort>
<fos gen field of kind> :: <aggregation kind> <field<name>

<field sort> = <sort>

Conditions on concrete syntax

∀*sd*∈<structure definition>: *sd.fieldNameList0.length* = |*sd.fieldNameList0.toSet*|

Each <field name> of a structure sort must be different from every other <field name> of the same <structure definition>.

Transformations

< <optional field>(<fields of sort>(*vis*, < *f* > ⌒ *rest*, *sort*))> **provided** *rest* ≠ *empty* =1=>
< <optional field>
(<fields of sort>(*vis*, < *f* >, *sort*)>, <optional field>(<fields of sort>(*vis*, *rest*, *sort*)))

< <mandatory field>(<fields of sort>(*vis*, < *f* > ⌒ *rest*, *sort*), *init*)> **provided** *rest* ≠ *empty* =1=>
< <mandatory field>(<fields of sort>(*vis*, < *f* >, *sort*), *init*),
<mandatory field>(<fields of sort>(*vis*, *rest*, *sort*), *init*) >

A <field list> containing a <field> with a list of <field name>s in a <fields of sort> is derived concrete syntax where this <field> is replaced by a list of <field>s separated by <end>, such that each <field> in this list resulted from copying the original <field> and substituting one <field name> for the list of <field name>s, in turn for each <field name> in the list.

b = <data type definition body>
(*entities*,
s = <structure definition>(*, *fields*),
<operations>(<operation signatures>(<operator list>(*operators*), *refs*, *defs*), *init*)
)

provided ¬ *b.parentAS0.identifier0.implicitSignaturesAdded*
=5=>
let *sort* = b.*parentAS0.identifier0* **in**
let *newoperators* =
<
<operation signature>(<name>("Make"),
< <argument>(**in**,*s1*) | *s1* **in** *s.fieldSortList0* ∧

```
                    ¬isOptionalField0(s1) ∧ defaultValue(s1) = undefined
            >,
            <result>(sort), empty)
    >
    in let newmethods =
        <   <operation signature>(
                <name>(fields.fieldNameList0[n].s-TOKEN + "Modify"),
                < <argument>(in, fields.fieldSortList0[n]) >,
                <result>( sort),empty) | n in 1..fields.fieldNameList0.length > ⌒
        <   <operation signature>(
                <name>(fields.fieldNameList0[n].s-TOKEN + "Extract"),
                < >,
                <result>( fields.fieldSortList0[n]),empty) | n in 1..fields.fieldNameList0.length > ⌒
        <   <operation signature>(
                <name>(n.s-TOKEN + "Present"),
                < >,
                <result>(<identifier>(<qualifier>(<name>("Predefined")), "Boolean")), empty)
            | n in fields.fieldNameList0: n.isOptionalField0 >
    in
        <data type definition>(entities, s, <operations>(<operation signatures>(
            <operator list>(operators ⌒ newoperators),
            refs, defs), init)
    endlet
and
    b.parentAS0.identifier0.implicitSignaturesAdded:= true
```

A structure definition is derived syntax for the definition of:

a) an operator Make to create structures;

b) methods to modify structures and to access component data items of structures; and

c) methods to test for the presence of optional component data items in structures.

The <arguments> for the Make operator contains the list of <field sort>s occurring in the field list in the order in which they occur. The result <sort> for the Make operator is the sort identifier of the structure sort. The Make operator creates a new structure and associates each field with the result of the corresponding formal parameter. If the actual parameter was omitted in the application of the Make operator, the corresponding field gets no value; that is, it becomes "undefined".

A <structure definition> introducing a sort named S implies a set of Dynamic-operation-signatures equivalent to the explicit definitions in the following <method list>, for each <field> in its <field list>:

virtual field-modify-operation-name (<field sort>) -> S;
virtual field-extract-operation-name -> <field sort>;
field-presence-operation-name -> Boolean;

where Boolean is the predefined Boolean sort, and <field sort> is the sort of the field.

The name of the implied method to modify a field, field-modify-operation-name, is the field name concatenated with "Modify". The implied method to modify a field associates the field with the result of its argument Expression. When <field sort> was an <anchored sort>, this association takes place only if the dynamic sort of the argument Expression is sort compatible with the <field sort> of this field. Otherwise, the predefined exception UndefinedField is raised.

The name of the implied method to access a field, field-extract-operation-name, is the field name concatenated with "Extract". The method to access a field returns the data item associated with that field. If, during interpretation, a field of a structure is "undefined", then applying the method to access this field to the structure leads to the raising of the predefined exception UndefinedField.

The name of the implied method to test for the presence of a field data item, field-presence-operation-name, is the field name concatenated with "Present". The method to test for the presence of a field data item returns the predefined Boolean value false if this field is "undefined", and the predefined Boolean value true otherwise. A method to test for the presence of a field data item is only defined if this <field> contained the keyword optional.

<mandatory field>(*fields*, *init*) **provided** *init* ≠ *undefined*
=> <optional field>(*fields*)

If a <field> is defined with a <field default initialization>, this is derived syntax for the definition of this <field> as optional.

<operator application>(*ident*, params = *first* ⌒ < *param* > ⌒ *last*)
provided *ident*.**s-**<operation name>.**s-***TOKEN* = "Make" ∧
param = *undefined* ∧
∃ *cons* ∈ <structure definition>: *cons.parentAS0*. *parentAS0* =
parentAS0ofKind(*ident.refersto0*, {<data type definition>})
∧ *cons.fieldNameList0*[first.*length* + 1].*defaultValue* ≠ *undefined*
=>
let *cons* = *take*({*cons* ∈ <structure definition>:
cons.parentAS0. *parentAS0* = *parentAS0ofKind*(*ident.refersto0*, {<data type definition>})} **in**
<method application>(<operator application>(ident, params),
<identifier>(*parentAS0ofKind*(*ident.refersto0*, {<data type definition>}.*qualifier0*,
<name>(*cons.fieldNameList0*[first.*length* + 1].s-*TOKEN* + "Modify")),
< *cons.fieldNameList0*[first.*length* + 1].*defaultValue* >)
endlet

When a structure of this sort is created and no actual argument is provided for the default field, an immediate modification of the field by the associated <constant expression> after structure creation is added.

Auxiliary functions

fieldNameList0(*d*:<structure definition>∪<choice definition>): <name>*=$_{def}$
if *d*∈ <structure definition> **then**
< *f*.**s-**<name> | *f* **in** *d*.**s-**<field>**-seq** >
else
< *f*.**s-**<name> | *f* **in** *d*.**s-**<choice of sort>**-seq** >
endif

fieldSortList0(*sd*: <structure definition>∪<choice definition>):<sort>*=$_{def}$
<*fn.parentAS0*.**s-**<sort> | *fn* **in** *sd.fieldNameList0*>

isOptionalField0(*n*: <name>): *BOOLEAN* =$_{def}$
case *n.parentAS0* **in**
| <optional field>(*) => *true*
| <mandatory field>(*, *) => *false*
otherwise
undefined
endif

defaultValue(*n*: <name>): <constant expression> =$_{def}$
case *n.parentAS0* **in**
| <mandatory field>(*, <field default initialization>(*e*)) => *e*
otherwise
undefined
endif

F2.2.9.6.3 Choice data types

Concrete syntax

<choice definition> :: [<visibility>] [<choice list>]

<choice list> = <choice of sort>+

<choice of sort> :: [<visibility>] <cos gen field of kind>+ <field sort>

<cos gen field of kind> :: <aggregation kind> <field<name>

Conditions on concrete syntax

∀*cd*∈<choice definition>: *cd.fieldNameList0.length* = |*cd.fieldNameList0.toSet*|

Each <field name> of a choice sort must be different from every other <field name> of the same <choice definition>.

Transformations

< <choice list>(<choice of sort>(*vis*, < *c* > ⌒ *rest*, *sort*))> **provided** *rest* ≠ *empty* =1=>
< <choice list>
(<choice of sort>(*vis*, < *c* >, *sort*)>, <choice list>(<choice of sort>(*vis*, *rest*, *sort*)))

< *d* = <data type definition>(*uses*, *preamble*, <data type heading>(*name*, *params*, *virt*), *spec*,
<data type definition body>(*entities*, <choice definition>(*visi*, *choices*), *ops*, *ini*)) >
=2=>
let *nn*= *newName* **in**
(**let** *emptyOperations* =
<operations>(<operation signatures>(<operator list>(*empty*), <method list>(*empty*)), *empty*)
in
let *anonStruct* =
<data type definition>(*uses*, *undefined*,
<data type heading>(*nn*, *params*, *virt*), *spec*,
<data type definition body>(*entities*,
<structure definition>(*visi*,
< <optional field>(
<fields of sort>(
c.**s-**<visibility>, *c*.**s-**<aggregation kind>, < *c*.**s-**<name> >, *c*.**s-**<sort>))
| *c* **in** *choices* >),
ops, *ini*))
in
let *anonLiterals* =
<data type definition>(*empty*, *undefined*,
<data type heading> (*nn* ⌒ "Present", *empty*, *undefined*), *undefined*,
<data type definition body>(*empty*, <literal list> (*undefined*, **unordered**,
< (< *c*.**s-**<name> | *c* **in** *choices* >) >,
emptyOperations, *undefined*)))
in
let *addOps* =
< <operation signature> (<operation preamble> (*undefined*, *undefined*), *c*.**s-**<name>,
< <arguments>(<argument> (*undefined*, *c*.**s-**<sort>) >),
<result> (*d.identifier*0)) | *c* **in** *choices* >
in
let *addMethods* =
< <operation signature> (<operation preamble> (**virtual**, *undefined*), *c*.**s-**<name> ⌒ "Modify",
< <arguments>(<argument> (*undefined*, *c*.**s-**<sort>) >),
<result> (*d.identifier*0)) | *c* **in** *choices* > ⌒
< <operation signature> (<operation preamble> (**virtual**, *undefined*), *c*.**s-**<name> ⌒ "Extract",
empty, <result> (*c*.**s-**<sort>)) | *c* **in** *choices* > ⌒
< <operation signature> (<operation preamble> (*undefined*, *undefined*), *c*.**s-**<name> ⌒ "Present",

```
            empty, <result> (“Boolean”)) | c in choices >
    in
    let presentOp =
        <operation signature> (<operation preamble> (undefined, undefined), “PresentExtract”,
            < <arguments>(<argument> (undefined, d.identifier0)) >,
            <result> (<identifier> (d.fullQualifier0, nn ⌒ “Present”)))
    in
    let addOpDefs =
        < <operation definition> (empty,
            <operation heading> (operator, <operation preamble> (undefined, undefined), empty, c.s-<name>,
                <<formal operation parameter>(undefined, in, <parameters of sort> (nn ⌒ “Par”, c.s-<sort>))
>,
                <operation result> (undefined, d.identifier0), empty), empty,
            <statement list> (empty,
                <return statement> (<return body> (<expression gen primary> (undefined,
                    <operator application> (nn ⌒ “Make”,
                        < <identifier> (c.s-<name>),
                        <operator application> (“Make”,
                            < (if par=c then <variable access> (nn ⌒ “Par”) else undefined endif )
                            | par in choices >)>))))))
        | c in choices >
    in
    let addMethodDefs =
        < <operation definition> (empty,
            <operation heading> (method, <operation preamble> (virtual, undefined), empty,
                c.s-<name> ⌒ “Modify”,
                <<formal operation parameter> (undefined, in, <parameters of sort> (nn⌒“Par”, c.s-<sort>))>,
                <operation result> (undefined, d.identifier0), empty), empty,
            <statement list> (empty,
                <return statement> (<return body> (<expression gen primary> (undefined,
                    <method application> (
                        <method application> (this, nn ⌒ “PresentModify”,
                            < <identifier> (empty, c.s-<name>) >),
                        nn ⌒ “ChoiceModify”,
                        < <operator application> (“Make”,
                            < (if par=c then <variable access> (nn ⌒ “Par”) else undefined endif )
                            | par in choices >)>))))))
        | c in choices > ⌒
        < <operation definition> (empty,
            <operation heading> (method, <operation preamble> (virtual, undefined), empty,
                c.s-<name> ⌒ “Extract”, empty,
                <operation result> (undefined, c.s-<sort>), empty), empty,
            <statement list> (empty,
                <return statement> (<return body> (<expression gen primary> (undefined,
                    <method application> (“Choice”, c.s-<name> ⌒ “Extract”,
                        < <variable access> (nn ⌒ “Par”) >))))))
        | c in choices > ⌒
        < <operation definition> (empty,
            <operation heading> (method, <operation preamble> (undefined, undefined), empty,
                c.s-<name> ⌒ “Present”, empty,
                <operation result> (undefined, “Boolean”), empty), empty,
            <statement list> (empty,
                <return statement> (<return body> (<expression gen primary> (undefined,
                    <method application> (“Choice”, c.s-<name> ⌒ “Present”,
```

```
                    < <variable access> (nn ⌒ "Par") >))))))
        | c in choices >
    in
    let presentOpDef =
        <operation definition> (empty,
            <operation heading> (method, <operation preamble> (undefined, undefined), empty,
                "PresentExtract",
                <<formal operation parameter>(undefined,in,<parameters of sort>(nn ⌒ "Par", d.identifier0))
>,
                <operation result> (undefined, <identifier> (d.fullQualifier0, nn ⌒ "Present")), empty), empty,
            <statement list> (empty,
                <return statement> (<return body> (<expression gen primary> (undefined,
                    <operator application> (nn ⌒"PresentExtract", < <variable access> (nn ⌒ "Par") >))))))
    in
        <data type definition>(uses, preamble, <data type heading>( name, params, virt), spec,
            <data type definition body>(entities⌒anonStruct⌒anonLiterals, undefined,
                <operations> (
                    <operation signatures> (
                        <operator list> (addOps),
                        <method list> (addMethods ⌒ < presentOp >)),
                    empty,
                    ops ⌒ addOpDefs ⌒ addMethodDefs ⌒ < presentOpDef >), ini))
    endlet // presentOpDef
    endlet // addMethodDefs
    endlet // addOpDefs
    endlet // presentOp
    endlet // addMethods
    endlet // addOps
    endlet // anonLiterals
    endlet // anonStruct
    endlet // emptyOperations
)
endlet // nn
```

A data type definition containing a <choice definition> is derived syntax and transformed in the following steps: Let Choice-name be the <data type name> of the original data type definition, then

a) A <value data type definition> with an anonymous name, anon, and a <structure definition> as the type constructor is added. In the <value data type definition>, for each <choice of sort>, a <field> is constructed containing the equivalent <fields of sort> with the keyword optional.

b) A <value data type definition> with an anonymous name, anonPresent, is added with a <literal list> containing all the <field name>s in the <choice list> as <literal name>s. The order of the literals is the same as the order in which the <field name>s were specified.

c) For each <choice of sort>, an <operation signature> is added to the <operator list> of operators operations representing an implied operator for creating data items:

   – field-name (field-sort) -> Choice-name;

   where field-name is the <field name> and field-sort is the <field sort> in <choice of sort>. The implied operator for creating data items creates a new structure by calling anonMake, initializing the field Choice with a newly created structure initialized with <field name>, and assigning the literal corresponding to the <field name> to the field Present.

d) For each <choice of sort>, an <operation signature>s are added to the <method list> of operators operations representing implied methods for modifying and accessing data items:

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

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

– field-present -> Boolean;

where field-extract is the name of the method implied by anon to access the corresponding field, field-modify is the name of the implied method implied by anon to modify that field, and field-present is the implied name of the method implied by anon to test for the presence of a field data item. Calls to field-extract and field-present are forwarded to Choice. Calls to field-modify assign a newly created structure initialized with <field name> to Choice and assign the literal corresponding to the <field name> to Present.

e) An <operation signature> is added to the <operator list> of operators operations representing an implied operator for obtaining the sort of the data item currently present in Choice:

– PresentExtract (Choice-name) -> anonPresent;

PresentExtract returns the value associated with the Present field.

f) anon and anonPresent are added to the <entity in data type>s of the original <data type definition>, and the operations and methods defined in the above steps are added to its <operations>.

F2.2.9.7 Behaviour of operations

Concrete syntax

<operation definition item> =
<operation definition> | <operation reference> | <external operation definition>

<operation reference> :: <operation kind> <operation signature>

<external operation definition> :: <operation kind> <operation signature>

<operation definition> ::
<package use clause>* <operation heading> <entity in operation>*
{ <operation body> | <statements> }

<operation heading> ::
<operation kind> <operation preamble> [<qualifier>] <operation name>
<formal operation parameters> [<operation result>]

<formal operation parameters> = <formal variable parameters>*

<operation kind> :: **operator** | **method**

<operation identifier> :: [<qualifier>] <operation name>

<entity in operation> =
<data definition>
| <variable definition>
| <select definition>

<operation body> ::
<start> {<free action>}*

<operation result> :: <result aggregation> [<variable<name>] <sort>

Conditions on concrete syntax

$\forall$ *opRef* $\in$ <operation reference>:
(*opRef*.**s-**<operation heading> .**s-**<formal variable parameters>**-seq** = *empty* $\vee$
opRef.**s-**<operation heading> .**s-**<operation result> = *undefined*) $\Rightarrow$
($\forall$ *opRef1* $\in$ <operation reference>:
opRef1 $\neq$ *opRef* $\wedge$ *opRef.parentAS0*= *opRef1.parentAS0* $\Rightarrow$ *opRef1.name0* $\neq$ *opRef.name0*)

$\forall$ *opDef* $\in$ <external operation definition>:
(*opDef*.**s-**<operation heading> .**s-**<formal variable parameters>**-seq** = *empty* $\vee$
opDef.**s-**<operation heading> .**s-**<operation result> = *undefined*) $\Rightarrow$
($\forall$ *opDef1* $\in$ <external operation definition>:
opDef1 $\neq$ *opDef* $\wedge$ *opDef.parentAS0*= *opDef1.parentAS0* $\Rightarrow$ *opDef1.name0* $\neq$ *opDef.name0*)

<formal operation parameters> and <operation result> in <operation reference> and <external operation definition> may be omitted if there is no other <operation reference> or <external operation definition>, respectively, within the same sort which has the same name.

∀*od*∈<operation definition>: ∃*os*∈<operation signature>:
od.name0 = *os.name0* ∧ *od.parentAS0*= *os.parentAS0* ⇒ *isSameOperationAndSignature0*(*od, os*)

For each <operation definition> if there exists an <operation signature> in the same scope unit having the same <operation name>, then it must have the same <argument sort>s and <parameter kind>s as specified in the <formal operation parameters> (if present) and the same <result sort> as specified in <operation result> (if present).

∀*os*∈<operation signature>: ∃!*od*∈<operation definition>:
od.parentAS0= *os.parentAS0*∧ *od.name0*= *os.name0*∧ *isSameOperationAndSignature0*(*od, os*)

For each <operation signature> at most one corresponding <operation definition> can be given.

∀*bs*∈<operation body>∪<statement>: *parentAS0ofKind*(*bs*, <operation definition>)≠*undefined* ⇒
(¬∃*ie*∈<imperative expression>: *isAncestorAS0*(*bs, ie*))∧
(∀*id*∈<identifier>: *id.idKind0*∉{**synonym**, **procedure**}∧*isAncestorAS0*(*bs, id*)⇒
isDefinedIn0(*getEntityDefinition0*
(*id, id.idKind0*), *parentAS0ofKind*(*bs*, <operation definition>)))

<operation body> as well as the <statement>s in <operation definition> may contain neither an <imperative expression> nor an <identifier> defined outside the enclosing <operation definition>, except for <synonym identifier>s, <operation identifier>s, <literal identifier>s and <sort>s.

Transformations

∀*bs*∈<operation body>∪<statement>: *parentAS0ofKind*(*bs*, <operation definition>)≠*undefined* ⇒
(¬∃*ie*∈<imperative expression>: *isAncestorAS0*(*bs, ie*))∧
(∀*id*∈<identifier>: *id.idKind0*∉{**synonym**, **procedure**}∧*isAncestorAS0*(*bs, id*)⇒
isDefinedIn0
(*getEntityDefinition0*(*id, id.idKind0*), *parentAS0ofKind*(*bs*, <operation definition>)))

For every <operation definition> which does not have a corresponding <operation signature>, an <operation signature> is constructed.

let *nn* = *newName* **in**
od = <operation definition>(*use*, <operation heading>(*kind*, *, *, *name*, *, *params*,
<operation result>(*var*, *sort*)), *entities*, *body*)
provided *od.operatorProcedureName* = *undefined*
=>
od
and
od.getEntities
=> *od.getEntities* ⌒
< <internal procedure definition>(*use*,
<procedure heading>(*undefined*, *undefined*, *nn*, *empty*, *undefined*, *undefined*,
(**if** *kind* = **method then**
< <formal variable parameters>(**inout**, <parameters of sort>(< thisname >,
parentAS0ofKind(od, <data type definition>).*identifier0*)) >
else < > **endif**) ⌒
< <formal variable parameters>(*p*.**s-**<parameter kind>, *p*.**s-**<parameters of sort>) |
p **in** *params* >,
<procedure result>(*var*, *sort*)),
entities, *makeProcedureBody*(*body*))>
and
od.operatorProcedureName:= *nn*

An <operation definition> is transformed into an <internal procedure definition>, having an anonymous name, having <procedure formal parameters> derived from the <formal operation

parameters>, and having a <result> derived from the <operation result>. The <procedure body> is derived from <operation body> if one was present, or, if the <operation definition> contains a <statements> list; the result of this transformation is an <internal procedure definition>. After the Model of <internal procedure definition> has been applied, the virtual start inserted by that Model is replaced by a start without <virtuality>.

The Procedure-definition corresponding to the resultant <internal procedure definition> is associated with the Operation-signature represented by the <operation signature>.

If the <operation definition> defines a method, then during the transformation into an <internal procedure definition> an initial parameter with <parameter kind> in/out is inserted into <formal operation parameters>, with the argument <sort> being the sort that is defined by the <data type definition> that constitutes the scope unit in which the <operation definition> occurs. The <variable name> in <formal operation parameters> for this inserted parameter is a newly formed anonymous name.

NOTE – It is not possible to specify an <operation definition> for a <literal signature>.

If any <operation definition> contains informal text, then the interpretation of expressions involving application of the corresponding operator or method is not formally defined by SDL-2010 but may be determined from the informal text by the interpreter. If informal text is specified, a complete formal specification has not been given in SDL-2010.

Auxiliary functions

isSameOperationAndSignature0(*od*: <operation definition>, *os*: <operation signature>): *BOOLEAN* =$_{\text{def}}$
let *seq1 = od.operationFormalparameterList0* **in**
let *seq2 = os.operationSignatureParameterList0* **in**
(*od*.**s-**<operation heading> .**s-**<operation result> ≠ *undefined* ⇒
isSameResult0(*od*.**s-**<operation heading> .**s-**<operation result>, *os*.**s-**<result>)∧
(*seq1*≠*empty* ⇒
seq1.*length* = *seq2*.*length*∧
(∀*i*∈1..*seq1*.*length*: *isSameSort0*(*seq1*[*i*].*parentAS0*.**s-**<sort>, *seq2*[*i*].**s-**<sort>) ∧
seq1[*i*].*parentAS0*.**s-**<parameter kind>= *seq2*[*i*].**s-**<parameter kind>)))
endlet

Get the list of formal parameters of an operation definition.

operationFormalparameterList0(*od*: <operation definition>): <name>* =$_{\text{def}}$
< *opl*.**s-**<parameters of sort>.**s-**<<name>**-seq** |
opl **in** *od*.**s-**<operation heading>.**s-**<formal variable parameters>**-seq** >

The following determines the entity kind of an <identifier> according to its position.

idKind0(*i*: <identifier>): *ENTITYKIND0* =$_{\text{def}}$
case *i.parentAS0* **of**
| <package use clause> => **package**
| <procedure reference> => **procedure**
| <system type reference> => **system type**
| <block type reference> => **block type**
| <process type reference> => **process type**
| <composite state type reference> => **state type**
| <textual interface gate definition> => **interface**
| <textual endpoint constraint> => *parentAS0ofKind*(*i, TYPEDEFINITION0*).*kind0*
| <agent type context parameter> => **agent type**
| <agent constraint atleast> => **agent type**
| <agent constraint exactly> => **agent type** | <procedure context parameter> => **procedure**
| <signal context parameter list> => **signal**
| <compositestate type context parameter> => **state type**
| <interface constraint> => **interface**
| <virtuality constraint> => *parentAS0ofKind*(*i, TYPEDEFINITION0*).*kind0*
| <procedure preamble> => **remote procedure**

| <channel endpoint> =>
if *getEntityDefinition0*(*i*, **agent**) ≠ *undefined* **then agent**
else state
endif
| <channel to channel connection> => **channel**
| <signal list item> ∪ <stimulus> ∪ <gate constraint> ∪ <valid input signal set>
∪ <channel path> ∪ <signal list definition> ∪ <interface use list> ∪ <save part> =>
if *getEntityDefinition0*(*i*, **signal**) ≠ *undefined* **then signal**
elseif *getEntityDefinition0*(*i*, **signallist**) ≠ *undefined* **then signallist**
elseif *getEntityDefinition0*(*i*, **timer**) ≠ *undefined* **then timer**
elseif *getEntityDefinition0*(*i*, **remote procedure**) ≠ *undefined* **then remote procedure**
elseif *getEntityDefinition0*(*i*, **remote variable**) ≠ *undefined* **then remote variable**
else interface
endif
| <remote procedure call body> => **remote procedure**
| <import expression> => **remote variable**
| <export statement> => **variable**
| <inner entry point> => **state**
| <outer exit point> => **state**
| <create body> =>
if *getEntityDefinition0*(*i*, **agent**) ≠ *undefined* **then agent**
else agent type
endif
| <procedure call body> => **procedure**
| <output body> => **signal**
| <via path> =>
if *getEntityDefinition0*(*i*, **channel**) ≠ *undefined* **then channel**
else gate
endif
| <destination> => **timer**
| <loop variable indication> => **variable**
| <set clause> => **timer**
| <reset clause> => **timer**
| <range check constrained sort> => **sort**
| <variables of sort gen name> => **remote variable**
| <timer active expression> => **timer**
| <type expression> => *parentAS0ofKind*(*i*, *TYPEDEFINITION0*).*kind0*
| *a*=<actual context parameter> =>
take({*f* ∈ <formal context parameter>: *isContextParameterCorresponded0*(*a*, *f*) }).*entityKind0*
| <communication constraints> => **timer**
| <anchored sort> ∪ <expanded sort> ∪ <formal parameter>
∪ <variable context parameter list> ∪ <remotevariable context parameter list> ∪ <sort constraint>
∪ <parameters of sort> ∪ <procedure result> ∪ <remote variable definition>
∪ <local variables of sort> ∪ <loop variable definition> ∪ <interface variable definition>
∪ <result> ∪ <field> ∪ <formal variable parameters> ∪ <operation result>
∪ <internal synonym definition item> ∪ <external synonym definition item>
∪ <range check expression> ∪ <variables of sort> ∪ <any expression>
∪ <synonym context parameter list> ∪ <syntype definition syntype> => **sort**
| <method application> => **method**
| <operator application> => **operator**
| <indexed primary> ∪ <field primary> ∪ <actual context parameter>
∪ <operand5> =>
if *getEntityDefinition0*(*i*, **variable**) ≠ *undefined* **then variable**
elseif *getEntityDefinition0*(*i*, **synonym**) ≠ *undefined* **then synonym**
else literal
endif
| <stimulus> => **variable**
| <assignment> => **variable**
| <indexed variable> => **variable**
| <field variable> => **variable**
endcase

makeProcedureBody(b: <operation body> ∪ <statements>) <procedure body> ∪ <statement>* =$_{def}$
case *b* **of**
| <operation body>(*onexc*, *start*, *actions*) => <procedure body>(*onexc*, *start*, *actions*)
| <statements>(*, *) => <compound statement>(*b*)
otherwise
undefined
endcase

F2.2.9.8 Additional data definition constructs

F2.2.9.8.1 Name class mapping

Name class mapping is defined in clause A.2.10 of [ITU-T Z.104] and is intended for use only to define use SDL-2010 language features and is not part of the SDL-2010 language itself. Whether it should continue to be formally described in Annex F needs further study.

Concrete syntax

<spelling term> :: <operation<name>

Transformations

A name class mapping is shorthand for a set of <operation definition>s or a set of <operation diagram>s. The set of <operation definition>s is derived from an <operation definition> by substituting each name in the equivalent set of names of the corresponding <name class operation> for each occurrence of <operation name> in the <operation definition>. The derived set of <operation definition>s contains all possible <operation definition>s that can be generated in this way. The same procedure is followed for deriving a set of <operation diagram>s.

The derived <operation definition>s and <operation diagram>s are considered legal even though a <string name> is not allowed as an <operation name> in the concrete syntax.

The derived <operation definition>s are added to <operation definitions> (if any) in the same <data type definition>. The derived <operation diagram>s are added to the list of diagrams where the original <operation definition> had occurred.

If an <operation definition> or <operation diagram> contains one or more <spelling term>s, each <spelling term> is replaced with a Charstring literal.

If, during the above transformation, the <operation name> in the <spelling term> had been replaced by an <operation name>, the <spelling term> is shorthand for a Charstring derived from the <operation name>. The Charstring contains the spelling of the <operation name>.

If, during the above transformation, the <operation name> in the <spelling term> had been replaced by a <string name>, the <spelling term> is shorthand for a Charstring derived from the <string name>. The Charstring contains the spelling of the <string name>.

F2.2.9.8.2 Restricted visibility

Concrete syntax

<visibility> = **public** | **protected** | **private**

Conditions on concrete syntax

∀*d*∈<data type definition>: *d.specialization0*≠ *undefined* ⇒
(∀*cons*∈<data type constructor>:
cons.surroundingScopeUnit0 = *d* ⇒ *cons*.**s-**<visibility> = *undefined*)

<visibility> must not precede a <literal list>, <structure definition>, or <choice definition> in a <data type definition> containing <data type specialization>.

∀*os*∈<operation signature>:
os.virtuality0 ∈{**redefined**, **finalized**}⇒*os.visibility0=undefined*

<visibility> must not be used in an <operation signature> that redefines an inherited operation signature.

Transformations

<operation preamble>(*virt*, *vis*) **provided** *vis* ≠ *undefined*
=><operation preamble>(*virt*, *undefined*)

<literal list>(*vis*, *sigs*) **provided** *vis* ≠ *undefined*
=> <literal list>(*undefined*, *sigs*)

<structure definition>(*vis*, *fields*) **provided** *vis* ≠ *undefined*
=> <structure definition>(*undefined*, *fields*)

<fields of sort>(*vis*, *fields*, *sort*) **provided** *vis* ≠ *undefined* => <fields of sort>(*undefined*, *fields*, *sort*)

<choice definition>(*vis*, *fields*) **provided** *vis* ≠ *undefined* => <choice definition>(*undefined*, *fields*)

<choice of sort>(*vis*, *fields*, *sort*) **provided** *vis* ≠ *undefined*
=> <choice of sort>(*undefined*, *fields*, *sort*)

If a <literal signature> or <operation signature> contains the keyword **public** in <visibility>, this is derived syntax for a signature having no protection.

Auxiliary functions

The function *isPrivate0* determines if a <literal signature> or an <operation signature> is private.

isPrivate0 (*s*: <literal signature> ∪ <operation signature>): *BOOLEAN* =$_{def}$
(*s.visibility0* = **private**)

The function *isPublic0* determines if a <literal signature> or an <operation signature> is public.

isPublic0 (*s*: <literal signature> ∪ <operation signature>): *BOOLEAN* =$_{def}$
(*s.visibility0* ≠ **private**) ∧ (*s.visibility0* ≠ **protected**)

F2.2.9.8.3 Syntypes

Abstract syntax

Syntype-definition	::	*Syntype-name* *Parent-sort-identifier* *Range-condition* [*Default-initialization*]
Parent-sort-identifier	=	*Sort-identifier*

Concrete syntax

<syntype> = <syntype<identifier>

<syntype definition> ::
<package use clause>*
{ <syntype definition syntype> | <syntype definition data type> }

<syntype definition syntype> ::
<syntype<name> <parent sort identifier>
[<default initialization>] [<constraint>]

<syntype definition data type> ::
<type preamble> <data type heading> [<data type specialization>]
<data type definition body> <constraint>

<parent sort identifier> = <sort>

Transformations

let *nn* = *newName* **in**
< <syntype definition>(*uses*,
<syntype definition data type>(*preamble*,
<data type heading>(*kind*, *name*, *params*, *vconstr*), *spec*, *body*, *constr*)) >
=>
< <syntype definition>(*uses*,
<syntype definition syntype>(name, <identifier>(*empty*, nn), *undefined*, *constr*)),
<data type definition>(*uses*, *preamble*, <data type heading>(*kind*, *nn*, *params*, *vconstr*),
spec, *body*) >

A <syntype definition> with the keywords value type or object type can be distinguished from a <data type definition> by the inclusion of a <constraint>. Such a <syntype definition> is shorthand for introducing a <data type definition> with an anonymous name followed by a <syntype definition> with the keyword **syntype** based on this anonymously named sort and including <constraint>.

Mapping to abstract syntax

| <syntype definition>(*, <syntype definition syntype>(*name*, *parent*, *, *constr*)) =>
mk-*Syntype-definition*(*Mapping*(*name*), *Mapping*(*parent*), *Mapping*(*constr*))

F2.2.9.8.4 Constraint

Abstract syntax

Range-condition	::	*Condition-item*-**set**
Condition-item	=	*Open-range* | *Closed-range* | *Size-constraint*
Open-range	::	*Operation-identifier Constant-expression*
Closed-range	::	*Constant-expression Constant-expression*
Size-constraint	::	*Operation-identifier* { *Open-range* | *Closed-range* }*

Concrete syntax

<constraint> = <range condition> | <size constraint>

<range condition> = <range>+

<range> = <closed range> | <open range>

<open range> = <constant> | <open range with operator>

<open range with operator> ::
{ <equals sign> | <not equals sign> | <less than sign> | <greater than sign> |
<less than or equals sign> | <greater than or equals sign> } <constant>

<constant> = <constant expression>

<closed range> :: <constant> <constant>

<size constraint> :: <range condition>

Conditions on concrete syntax

∀*sd*∈<syntype definition>: *sd*.**s-implicit**.**s-**<constraint>∈<range condition> ⇒
(**let** *rc* = *sd*.**s-implicit**.**s-**<constraint> **in**
(∀*sym*∈<less than sign>: *isAncestorAS0*(*rc*, *sym*)⇒ *isDefinedSym0*(*sd*, "<"))∧
(∀*sym*∈<greater than sign>: *isAncestorAS0*(*rc*, *sym*)⇒ *isDefinedSym0*(*sd*, ">"))∧
(∀*sym*∈<less than or equals sign>: *isAncestorAS0*(*rc*, *sym*)⇒ *isDefinedSym0*(*sd*, "<="))∧
(∀*sym*∈<greater than or equals sign>: *isAncestorAS0*(*rc*, *sym*)⇒ *isDefinedSym0*(*sd*, ">="))
endlet)

The symbol "<" must only be used in the concrete syntax of the <range condition> if that symbol has been defined with an <operation signature>: "<" (P, P) -> <<**package** Predefined>>Boolean; where P is the sort of the syntype, and similarly for the symbols ("<=", ">", ">=", respectively).

∀*sd*∈<syntype definition>: *sd*.**s-implicit**.**s-**<constraint>∈<range condition>⇒
∀*cr*∈<closed range>:
isAncestorAS0(*sd*.**s-implicit**.**s-**<range condition>,*cr*)⇒ *isDefinedSym0*(*sd*, "<=")

A <closed range> must only be used if the symbol "<=" is defined with an <operation signature>: "<=" (P, P) -> <<**package** Predefined>>Boolean; where P is the sort of the syntype.

∀*sd*∈<syntype definition>:∀*rc*∈<range condition>:∀*ce*∈<constant expression>:
isAncestorAS0(*rc*, *ce*)∧*rc.surroundingScopeUnit0=sd*⇒*isSameSort0*(*ce.staticSort0*, *sd.identifier0*)

A <constant expression> in a <range condition> must have the same sort as the sort of the syntype.

∀*sd*∈<syntype definition>: ∀*sc*∈<size constraint>:
sc=*sd*.**s-implicit**.**s-**<constraint>⇒*isDefinedSym0*(*sd*, "Length")

A <size constraint> must only be used in the concrete syntax of the <range condition> if the symbol Length has been defined with an <operation signature>: Length (P) -> <<**package** Predefined>> Natural; where P is the sort of the syntype.

Mapping to abstract syntax

| r = <range condition>(*items*) =>
mk-*Range-condition*(*toSet*(
< **if** *item* ∈ <constant> **then mk-***Open-range*(*rangeOperator*(r, "="), *Mapping*(*item*))
else *Mapping*(item) **endif** | *item* **in** *items* >)
| *r* = <open range with operator>(<equals sign>(), *const*) =>
mk-*Open-range*(*rangeOperator*(r, "="), *Mapping*(*const*))
| *r* = <open range with operator>(<not equals sign>(), *const*) =>
mk-*Open-range*(*rangeOperator*(r, "/="), *Mapping*(*const*))
| *r* = <open range with operator>(<less than sign>(), *const*) =>
mk-*Open-range*(*rangeOperator*(r, "<"), *Mapping*(*const*))
| *r* = <open range gen with operator>(<greater than sign>(), *const*) =>
mk-*Open-range*(*rangeOperator*(r, ">"), *Mapping*(*const*))
| *r* = <open range with operator>(<less than or equals sign>(), *const*) =>
mk-*Open-range*(*rangeOperator*(r, "<="), *Mapping*(*const*))
| *r* = <open range with operator>(<greater than or equals sign>(), *const*) =>
mk-*Open-range*(*rangeOperator*(r, ">="), *Mapping*(*const*))
| <closed range>(*c1*, *c2*) =>
mk-*Closed-range*(**mk-***Open-range*(*rangeOperator*(r, ">="), *Mapping*(*c1*),
mk-*Open-range*(*rangeOperator*(r, "<="), *Mapping*(*c2*)) }

Auxiliary functions

The function *isPredefSort0* is used to determine the predefined sorts.

isPredefSort0(*s*: <sort>): *BOOLEAN* =$_{def}$
getEntityDefinition0(*s*, **sort**)=*undefined*∧*s*.**s-**<name>∈*PREDEFINEDSORT0*

The function *isDefinedSym0* is used to determine if the given symbol is defined and the each parameter's sort is the same as that of the specified syntype.

isDefinedSym0(*sd*:<syntype definition>, *sym*: *SYMBOL0*): *BOOLEAN* =$_{def}$
(**let** *dtd*=*sd.derivedDataType0* **in**
if *sym*∈{"<",">","<=",">="} **then**
(∃*ll*∈<literal list>: *ll.surroundingScopeUnit0 = dtd*)∨
(∃*os*∈<operation signature>: (*os.surroundingScopeUnit0=dtd*)∧
(**let** *fpl*= *os.operationSignatureParameterList0* **in**
os.entityName0= *sym* ∧
isPredefSort0(*os*.**s-**<result>.**s-**<sort>) ∧ *os*.**s-**<result>.s-<sort>.**s-**<name>= "Boolean" ∧

fpl.*length* = 2 ∧
getEntityDefinition0(*fpl*[1].**s-**<formal parameter>.**s-**<sort>, **sort**) =*sd*∧
getEntityDefinition0(*fpl*[2].**s-**<formal parameter>.**s-**<sort>, **sort**) =*sd*
endlet))
else // *sym*∈{"Length"}
(∃*os*∈<operation signature>: *os*.*surroundingScopeUnit0=dtd*∧
(**let** *fpl=os*.*operationSignatureParameterList0* **in**
os.*name0*= "Length"∧
isPredefSort0(*os*.**s-**<result>.**s-**<sort>) ∧ *os*.**s-**<result>.s-<sort>.**s-**<name>= "Natural"∧
fpl.*length*= 1∧
getEntityDefinition0
(*fpl*[1].**s-**<formal parameter>.**s-**<sort>, **sort**) *derivedDataType0=dtd*
endlet))
endlet)

rangeOperator(t: *TOKEN*): *Identifier*

F2.2.9.8.5 Synonym definition

Concrete syntax

<synonym definition> :: <synonym definition item>+

<synonym definition item> =
<internal synonym definition item> | <external synonym definition item>

<internal synonym definition item> ::
<synonym<name> [<sort>] <constant expression>

<external synonym definition item> ::
<synonym<name> <predefined<sort>

Conditions on concrete syntax

∀*syno*∈<internal synonym definition item>:
¬ *isContainedInConsExp0*(*syno*, *syno*.**s-**<constant expression>)

The <constant expression> must not refer to the synonym defined by the <synonym definition> either directly or indirectly (via another synonym).

∀*sdi*∈<internal synonym definition item>:*sdi*.**s-**<sort> ≠ *undefined*⇒
∃*s*∈<sort>: *s*∈*sdi*.**s-**<constant expression>.*staticSortSet0*∧*isSameSort0*(*s*, *sdi*.**s-**<sort>)

If a <sort> is specified, the result of the <constant expression> has a static sort of <sort>. It must be possible for <constant expression> to have that sort.

∀ *sdi*∈<internal synonym definition item>:
|*sdi*.**s-**<constant expression>.*staticSortSet0*|>1⇒ *sdi*.**s-**<sort> ≠ *undefined*

If the sort of the <constant expression> cannot be uniquely determined, then a sort must be specified in the <synonym definition>.

Auxiliary functions

The function *isContainedInConsExp0* is used to determine if a <constant expression> refers to the synonym defined by the enclosing <synonym definition> either directly or indirectly.

isContainedInConsExp0(*def*: <internal synonym definition item>, *exp*: <constant expression>):
BOOLEAN =def
∃*synoId*∈<synonym>: *isAncestorAS0*(*exp*, *synId*) ∧
(*def* = *getEntityDefinition0*(*synoId*, **synonym**)∨
isContainedInConsExp0
(*def*, *getEntityDefinition0*(*synoId*, **synonym**).**s-**<constant expression>))

F2.2.9.9 Use of data

F2.2.9.9.1 Expressions and expressions as actual parameters

Abstract syntax

Expression	=	*Constant-expression*
	|	*Active-expression*
Constant-expression	=	*Literal*
	|	*Conditional-expression*
	|	*Equality-expression*
	|	*Operation-application*
	|	*Range-check-expression*
	|	*Type-check-expression*
	|	*Type-coercion*
Active-expression	=	*Variable-access*
	|	*Conditional-expression*
	|	*Operation-application*
	|	*Equality-expression*
	|	*Imperative-expression*
	|	*Range-check-expression*
	|	*Type-check-expression*
	|	*Type-coercion*
	|	*Value-returning-call-node*
	|	*Encoding-expression*
	|	*Decoding-expression*
Imperative-expression	=	*Now-expression*
	|	*Pid-expression*
	|	*Timer-active-expression*
	|	*Timer-remaining-duration*
	|	*Active-agents-expression*
	|	*Any-expression*
	|	*State-expression*
Actual-parameters	::	{ *Expression* | UNDEFINED }*

Please note that the above definition could be simplified. This can be done by omitting the difference between active expressions and constant expressions. This difference does not show up at any place, so it could be simply dropped.

Concrete syntax

<expression> =
    <expression0>
  | <range check expression>
  | <type coercion>

<expression0> =
    <binary expression>
  | <operand5>
  | <equality expression>
  | <create expression>
  | <value returning procedure call>
  | <decoding expression>
  | <encoding expression>

<simple expression> = <constant expression>

<actual parameters> = <actual parameter list>

<actual parameter list> =[<actual parameter>]+

<actual parameter> = <expression> | <undefined>

<constant expression> = <constant<expression>

<binary expression> ::
<expression> <infix operation name> <expression>

<operand5> ::[<hyphen> | **not**] <primary>

<primary> =
<operator application>
| <literal>
| <expression>
| <conditional expression>
| <extended primary>
| <active primary>
| <synonym>

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

<expression list> = <expression>+

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

Conditions on concrete syntax

∀*consExp*∈<expression>: *isConstantExpression0*(*consExp*)⇒
¬∃*activePri*∈<active primary>: *isAncestorAS0*(*consExp, activePri*)

A <constant expression> must not contain an <active primary>.

∀*id*∈<identifier>: ∀*expr*∈<expression>:
isSimpleExpression0(*expr*)∧ *isAncestorAS0*(*expr*, *id*)∧*id.idKind0*∈{**literal**, **operator**, **method**}⇒
getEntityDefinition0(*id*, *id.idKind0*)∈*PREDEFINEDDEFINITION0*

A <simple expression> must contain only literals, operators, and methods defined within the **package** Predefined, as defined in Annex D of Recommendation ITU-T Z.100.

Transformations

<binary expression>(*x*,<implies sign>,*y*) =8=> <operator application>("=>",<*x*,*y*>)
<binary expression>(*x*,**or**,*y*) =8=> <operator application>("or",<*x*,*y*>)
<binary expression>(*x*,**xor**,*y*) =8=> <operator application>("xor",<*x*,*y*>)
<binary expression>(*x*,**and,***y*) =8=> <operator application>("and",<*x*,*y*>)
<binary expression>(*x*,<greater than sign>,*y*) =8=> <operator application>(">",<*x*,*y*>)
<binary expression>(*x*,<greater than or equals sign>,*y*) =8=> <operator application>(">=",<*x*,*y*>)
<binary expression>(*x*,<less than sign>,*y*) =8=> <operator application>("<",<*x*,*y*>)
<binary expression>(*x*,<less than or equals sign>,*y*) =8=> <operator application>("<=",<*x*,*y*>)
<binary expression>(*x*,**in,***y*) =8=> <operator application>("in",<*x*,*y*>)
<binary expression>(*x*,<plus sign>,*y*) =8=> <operator application>("+",<*x*,*y*>)
<binary expression>(*x*,<hyphen>,*y*) =8=> <operator application>("-",<*x*,*y*>)
<binary expression>(*x*,<concatenation sign>,*y*) =8=> <operator application>("//",<*x*,*y*>)
<binary expression>(*x*,<asterisk>,*y*) =8=> <operator application>("*",<*x*,*y*>)
<binary expression>(*x*,<solidus>,*y*) =8=> <operator application>("/",<*x*,*y*>)
<binary expression>(*x*,**mod**,*y*) =8=> <operator application>("mod",<*x*,*y*>)
<binary expression>(*x*,**rem**,*y*) =8=> <operator application>("rem",<*x*,*y*>)
<operand5>(<hyphen>,*x*) =8=> <operator application>("-",<*x*>)
<operand5>(**not**,*x*) =8=> <operator application>("not",<*x*>)

An expression of the form

<expression> <infix operation name> <expression>

is derived syntax for

<quotation mark> <infix operation name> <quotation mark> (<expression>, <expression>)
where <quotation mark> <infix operation name> <quotation mark> represents an Operation-name.

Similarly,

<monadic operation name> <expression>
is derived syntax for

<quotation mark> <monadic operation name> <quotation mark> (<expression>)
where <quotation mark> <monadic operation name> <quotation mark> represents an Operation-name.

Auxiliary functions

staticSort0(*expr*:<expression>):<sort>=$_{def}$
take(*expr.staticSortSet0*)

Determine if an <expression> is a <constant expression> according to its position.

isConstantExpression0(*expr*: <expression>): *BOOLEAN*=$_{def}$
(*expr.parentAS0* ∈
<default initialization>∪
<timer default initialization>∪
<field default initialization> ∪
<internal synonym definition item>∪
<open range>∪
<transition option>∪
<variables of sort>) ∨
isSimpleExpression0(*expr*)

Determine if an <expression> is a <simple expression> according to its position.

isSimpleExpression0(*expr*: <expression>): *BOOLEAN*=$_{def}$
expr.parentAS0∈<number of instances>∪<named number>

F2.2.9.9.2 Literal

Abstract syntax

Literal :: *Literal-identifier*

Concrete syntax

<literal> = <literal identifier>

<literal identifier> :: <qualifier> <literal name>

Mapping to abstract syntax

| <literal identifier>(*qual*, *name*) => **mk-***Literal-identifier*(*Mapping*(*qual*), *Mapping*(*name*))

F2.2.9.9.3 Synonym

Concrete syntax

<synonym> :: <synonym<identifier>

Transformations

<synonym>(*ident*)
provided *ident.refersto0* ∈ <internal synonym definition item>
=>
ident.refersto0 .**s-**<constant expression>

A <synonym> represents the <constant expression> defined by the <synonym definition> identified by the <synonym identifier>. An <identifier> used in the <constant expression> represents an Identifier in the abstract syntax according to the context of the <synonym definition>.

F2.2.9.9.4 Extended primary

Concrete syntax

<extended primary> =
    <indexed primary>
  | <field primary>
  | <composite primary>

<indexed primary> :: <primary> <actual parameter list>

<field primary> :: [<primary>] { <field name> | <field number> | <as signal> }

<field name> = <name>

<field number> = <Natural<name>

<composite primary> :: <qualifier> <actual parameter list>

Transformations

<indexed primary>(*prim*, *params*)
=8=> <method application>(*prim*, <identifier>(*empty*, "Extract"), *params*)

An <indexed primary> is derived concrete syntax for

<primary> <full stop> Extract (<actual parameter list>)

The abstract syntax is determined from this concrete expression according to clause 12.2.1 of [ITU-T Z.101].

<field primary>(*prim*, *field*)
provided prim ≠ *undefined*
=8=> <method application>(*prim*, *modifyExtractName*(*field*, "Modify"), *empty*)

A <field primary> is derived concrete syntax for

<primary> <full stop> field-extract-operation-name

where the field-extract-operation-name is formed from the concatenation of the field name and "Extract" in that order. The abstract syntax is determined from this concrete expression according to clause 12.2.1 of [ITU-T Z.101]. The transformation according to this model is performed before the modification of the signature of methods in clause 12.1.3 of [ITU-T Z.104].

<field primary>(*undefined*, *field*)
=8=> <field primary>(**THIS**, *field*)

When the <field primary> has the form <field name>, this is derived syntax for:

this ! <field name>

<composite primary>(*qual*, *params*)
=8=>
<operator application>(<identifier>(*qual*, "Make"), *params*)

A <composite primary> is derived concrete syntax for:

<qualifier> Make (<actual parameter list>)

if any actual parameters were present, or

<qualifier> Make

otherwise, and where the <qualifier> is inserted only if it was present in the <composite primary>. The abstract syntax is determined from this concrete expression according to clause 12.2.1 of [ITU-T Z.101].

F2.2.9.9.5 Equality expression

Abstract syntax

Equality-expression	=	*Positive-equality-expression* | *Negative-equality-expression*
Positive-equality-expression	::	*First-operand Second-operand*
Negative-equality-expression	::	*First-operand Second-operand*
First-operand	=	*Expression*
Second-operand	=	*Expression*

Concrete syntax

<equality expression> ::
    <expression> { <equals sign> | <not equals sign> } <expression>

Conditions on concrete syntax

∀*ee*∈<equality expression>:
  (**let** *set1* = *ee*.**s-**<expression>.*staticSortSet0* **in**
  let *set2* = *ee*.**s2-**<expression>.*staticSortSet0* **in**
    ∃*s1*∈*set1*:∃*s2*∈*set2*:*isSortCompatible0*(*s1*, *s2*)∨ *isSortCompatible0*(*s2*, *s1*)
  endlet)

An <equality expression> is legal concrete syntax only if the sort of one of its operand is sort compatible to the sort of the other operand.

Transformations

<equality expression>(*x*, <not equals sign>, *y*)
=8=> <operand5>(**not**, <equality expression>(*x*, <equals sign>, *y*))

Mapping to abstract syntax

| <equality expression>(*first*, <equals sign>, *second*)
  => **mk-***Equality-expression*(*Mapping*(*first*), *Mapping*(*second*))

F2.2.9.9.6 Conditional expression

Abstract syntax

Conditional-expression	::	*Boolean-expression* *Consequence-expression* *Alternative-expression*
Consequence-expression	=	*Expression*
Alternative-expression	=	*Expression*

Conditions on abstract syntax

∀*c* ∈ *Conditional-expression*:
  (*c*.**s-***Consequence-expression.staticSort1*= *c*.**s-***Alternative-expression.staticSort1*) ∧
  (*c*.**s-***Boolean-expression.staticSort1*= **mk-***Identifier*("Predefined", "Boolean"))

For any *Conditional-expression*, the sort of the *Consequence-expression* must be the same as that of the *Alternative-expression*, and the sort of a *Boolean-expression* must be *BOOLEAN*.

Concrete syntax

<conditional expression> ::
    <Boolean<expression> <consequence expression> <alternative expression>

<consequence expression> = <expression>

<alternative expression> = <expression>

Conditions on concrete syntax

∀*ce*∈<conditional expression>:
let *set1*=*ce*.**s-**<consequence expression>.*staticSortSet0* **in**
let *set2*=*ce*.**s-**<alternative expression>.*staticSortSet0* **in**
|*set1*| =1 ∧ |*set2*| =1∧ *isSameSort0*(*set1.take*, *set2.take*)
endlet
endlet

The sort of the <consequence expression> must be the same as the sort of the <alternative expression>.

Mapping to abstract syntax

| <conditional expression>(*e1*, *e2*, *e3*) =>

mk-*Conditional-expression*(*Mapping*(*e1*), *Mapping*(*e2*), *Mapping*(*e3*))

F2.2.9.9.7 Operation application

Abstract syntax

Operation-application :: *Operation-identifier Actual-parameters*

Conditions on abstract syntax

∀*oa* ∈ *Operation-application*:
let *os* = *getEntityDefinition1*(*oa*, **operation**) **in**
isActualAndFormalParameterMatched1(*oa*.**s-***Expression*-**seq**, *os.formalParameterSortList1*)
endlet

The *Operation-identifier* in the *Operation-application* must be visible. Each *Expression* in the list of *Expression*s after the *Operation-identifier* must be sort compatible to the corresponding (by position) sort in the list of *Formal-arguments* of the *Operation-signature*.

Concrete syntax

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

<operator application> :: <operation identifier> <actual parameters>

<method application> :: <primary> <operation identifier> <actual parameters>

Conditions on concrete syntax

∀*methodApp*∈<method application>:
getEntityDefinition0(*methodApp*.**s-**<identifier>, **method**) ≠ *undefined*

A <method application> is legal concrete syntax only if <operation identifier> represents a method.

Transformations

<method application>(*prim*, *ident*, *params*) =8=>
<operator application>(*ident*, < *prim* > ⌒ *params*)

The concrete syntax form

<expression> <full stop> <operation identifier> [<actual parameters>]

is derived concrete syntax for

<operation identifier> new-actual-parameters

where new-actual-parameters is <actual parameters> containing only <expression>, if <actual parameters> was not present; otherwise new-actual-parameters is obtained by inserting <expression> before the first optional expression in <actual parameters>.

Mapping to abstract syntax

<operator application>(*ident*, *params*) =>
mk-*Operation-application*(*Mapping*(*ident*),*Mapping*(*params*))

Auxiliary functions

Get the actual parameter list associated with the *Operation-identifier*.

actualParameterListOfOpId1(*id*: *Operation-identifier*): [*Expression*]*=$_{\text{def}}$
case *id.parentAS1* **of**
| *Open-range*=> <*id.parentAS1*.**s-***Constant-expression*>
| *Operation-application*=> < *exp* | *exp* **in** *id.parentAS1*.**s-***Expression*-**seq**>
endcase

F2.2.9.9.8 Range check expression

Abstract syntax

Type-check-expression	::	*Expression Parent-sort-identifier*
Range-check-expression	::	*Expression Parent-sort-identifier Range-condition*

Concrete syntax

<range check expression> ::
<expression>
{ <range check constrained sort> | <syntype> | <pid sort> | <datatype<type expression> }

<range check constrained sort> :: <sort<identifier> <constraint>

Further study needed of the use of <syntype> and <pid sort> (which were previously represented by <sort>).

Further study needed if the form <datatype<type expression> is used, in which case the <range check expression> represents a *Type-check-expression*.

Conditions on concrete syntax

∀*rcExpr*∈<range check expression>:
let *s* = *take*({*s*| ((*s*∈<sort>)∧(*s* = *rcExpr*.**s-implicit**))∨
((*s*∈<identifier>)∧(*s* = *rcExpr*.**s-implicit**. **s-**<identifier>))}) **in**
isSameSort0(*rcExpr*.**s-**<expression>.*staticSort0*, *s*)
endlet)

The sort of <operand2> must be the same as the sort identified by <sort identifier> or <sort>.

Transformations

<range check expression>(*i* = <identifier>(*,*))
=>
if *i.refersto0* ∈ <syntype definition> **then**
<range check expression>(<range check constrained sort>(*i*,
i.refersto0.s-<syntype definition syntype>.**s-**<constraint>)
else
<operand5>(*undefined*,
<literal>(<identifier>(< <path item>(**package**, <name>("Predefined"),
<path item>(**type**, <name>("Boolean") >),"true"))
endif

<range check expression>(*sort*)
provided *sort* ∈ <sort> \ <identifier>

=>
<operand5>(*undefined*,
<literal>(<identifier>(< <path item>(**package**, <name>("Predefined"),
<path item>(**type**, <name>("Boolean") >),"true"))

Specifying a <sort> is derived syntax for specifying the <constraint> of the data type that defined the <sort>. If that data type was not defined with a <constraint>, the <range check expression> is not evaluated and the <range check expression> is derived syntax for specifying the predefined Boolean value true.

Mapping to abstract syntax

| <range check expression>(*expr*, *ident*) =>
mk-*Range-check-expression*(*Mapping*(*expr*), *Mapping*(*ident*)
| <range check constrained sort>(*, *constraint*) => *Mapping*(*constraint*)

F2.2.9.9.9 Variable definition

Abstract syntax

Variable-definition	::	*Variable-name* *Sort-reference-identifier* *Aggregation-kind* [*Constant-expression*]
Aggregation-kind	=	**PART** | **REF**

Conditions on abstract syntax

∀*d* ∈ *Variable-definition*: *d*.**s-***Constant-expression*≠ *undefined* ⇒
d.**s-***Constant-expression*. *staticSort1*= *d*.**s-***Sort-reference-identifier*

If the *Constant-expression* is present, it must be of the same sort as the one denoted by *Sort-reference-identifier*.

Concrete syntax

<variable definition> :: <variables of sort>+ | <exported variables of sort>+

<variables of sort> ::
<aggregation kind> <variable<name>+ <sort> [<constant expression>]

<exported variables of sort> ::
<aggregation kind> <exported variable>+ <sort> [<constant expression>]

<exported variable> :: <variable<name> <remote variable<identifier>

<aggregation kind> :: [**part** | **ref**]

Further study is needed for <aggregation kind> and all the places it is used (not just variable definition). The <aggregation kind> has been added for <variables of sort> but there are many other places where it needs to be added. If <aggregation kind> is empty or **part** the *Aggregation-kind* is **PART**. If <aggregation kind> is **ref** the *Aggregation-kind* is **REF**. Needs to be written below as a Mapping.

Conditions on concrete syntax

∀*d*∈<agent definition> ∪<agent type definition>:
¬(∃*v1*,*v2*∈<exported variable>: *v1* ≠ *v2* ∧ *v1*.**s-**<identifier> = *v2*.**s-**<identifier> ∧
v1. *parentAS0*.*parentAS0*∈<variable definition> ∧
v2. *parentAS0*.*parentAS0*∈<variable definition>∧
v1.*surroundingScopeUnit0* = *d* ∧ *v2*.*surroundingScopeUnit0* = *d*)

Two exported variables in an agent cannot mention the same <remote variable identifier>.

Transformations

< <variable definition>(< *v* > ⌒ *rest*) > **provided** *rest* ≠ *empty* =1=>
< <variable definition> (< *v* >), <variable definition> (*rest*) >

< <variables of sort> (*ak*, < *v* > ⌒ *rest*, *sort*, *expr*) > **provided** *rest* ≠ *empty* =1=>
< <variables of sort> (*ak*, < *v* >, *sort*, *expr*), <variables of sort> (*ak*, *rest*, *sort*, *expr*) >

< <exported variables of sort> (*ak*, < *v* > ⌒ *rest*, *sort*, *expr*) > **provided** *rest* ≠ *empty* =1=>
< <exported variables of sort> (*ak*, < *v* >, *sort*, *expr*), <exported variables of sort> (*ak*, *rest*, *sort*, *expr*) >

A <variable definition> that defines multiple variables is a shorthand for a sequence of <variable definition>s, each defining one variable.

Mapping to abstract syntax

| <variable definition>(*, < *var* >) => *Mapping*(*var*)
| <variables of sort>(*ak*, <*name*>, *sort*, *const*)
=> **mk-***Variable-definition*(*Mapping*(*name*), *Mapping*(*sort*), *Mapping*(*ak*), *Mapping*(*const*))
| <exported variables of sort>(*ak*, <*ev*>, *sort*, *const*)
=> **mk-***Variable-definition*(*Mapping*(*ev*.**s-**<name>), *Mapping*(*sort*), *Mapping*(*ak*), *Mapping*(*const*))

F2.2.9.9.10 Variable access

Abstract syntax

Variable-access = *Variable-identifier*

Concrete syntax

<variable access> :: { <variable<identifier> | **this** | <import expression> }

Conditions on concrete syntax

∀*va*∈<variable access>: *va*.**s-implicit** = **this** ⇒
(*parentAS0ofKind*(*va*, <operation definition>) ≠ *undefined* ∧
parentAS0ofKind(*va*, <operation definition>).*kind0*=**method**)

this must only occur in method definitions.

Transformations

va = <variable access>(**this**)
=8=> <variable access>(*parentAS0ofKind*(va, <operation definition>).**s-**<operation heading>.
s-<formal variable parameters>.*head*.**s-**<parameters of sort>.**s-**<name>.*head*)

A <variable access> using the keyword **this** is replaced by the anonymous name introduced as the name of the leading parameter in <arguments> according to clause 12.1.83 of [ITU-T Z.104].

Mapping to abstract syntax

| <variable access>(*identifier*) => **mk-***Variable-identifier*(*Mapping*(*identifier*))

F2.2.9.9.11 Assignment

Abstract syntax

Assignment :: *Variable-identifier Expression*

Conditions on abstract syntax

∀*a* ∈ *Assignment*: ∃*d* ∈ *Variable-definition*:
(*d*=*getEntityDefinition1*(*a*. **s-***Variable-identifier*,**variable**)) ∧
isCompatibleTo1(*a*.**s-***Expression*.*staticSort1*, *d*.**s-***Sort-reference-identifier*)

In an *Assignment*, the sort of the *Expression* must be sort compatible to the sort of the *Variable-identifier*.

Concrete syntax

<assignment> :: <variable> <expression>

<variable> = <variable<identifier> | <extended variable>

Mapping to abstract syntax

| <assignment>(*var*,*expr*)
=> **mk-***Assignment*(*var*,*expr*)

F2.2.9.9.12 Extended variable

Concrete syntax

<extended variable> = <indexed variable> | <field variable>

<indexed variable> :: <variable> <actual parameter list>

<field variable> :: <variable> { <field name> | <field number> | <as signal> }

Transformations

<assignment>(<indexed variable>(*var*, *params*), *expr*)
=8=>
<assignment>(*var*, <method application>(*var*, <identifier>(*empty*, "Modify"), *expr*))

<indexed variable> is derived concrete syntax for

<variable> <is assigned sign> <variable> <full stop> Modify (expressionlist)

where expressionlist is constructed by appending <expression> to the <actual parameter list>. The abstract grammar is determined from this concrete expression according to clause 12.2.1 of [ITU-T Z.101]. The same model applies to the second form of <indexed variable>.

<assignment>(<field variable>(*var*, *fieldname*), *expr*)
=8=>
<assignment>
(*var*, <method application>(*var*, *modifyExtractName*(*fieldname*, "Modify"), *expr*))

The concrete syntax form

<variable> <exclamation mark> <field name> <is assigned sign> <expression>

is derived concrete syntax for

<variable> <full stop> field-modify-operation-name (<expression>)

where the field-modify-operation-name is formed from the concatenation of the field name and "Modify". The abstract syntax is determined from this concrete expression according to clause 12.2.1 of [ITU-T Z.101]. The same model applies to the second form of <field variable>.

Auxiliary functions

modifyExtractName(*name*: <identifier>, *suffix*: *TOKEN*):<identifier> =$_{def}$
mk-<identifier>(*name*.**s-**<qualifier>, *name*.**s-**<name> + *suffix*)

F2.2.9.9.13 Default initialization

Abstract syntax

Default-initialization = *Constant-expression*

Concrete syntax

<default initialization> ::[<virtuality>] [<constant expression>]

Conditions on concrete syntax

∀*def*∈<data type definition>: ¬∃*d1*,*d2*∈<default initialization>:
d1 ≠ *d2*∧*d1.surroundingScopeUnit0=def*∧*d2.surroundingScopeUnit0=def*

∀*sd*∈<syntype definition>:
(*sd*.**s-implicit**.**s-**<default initialization>≠*undefined*)⇒
¬∃*d*∈<default initialization>: *isDefinedIn0*(*d*, *sd.derivedDataType0*)

A <data type definition> or <syntype definition> must contain at most one <default initialization>.

∀*init*∈<default initialization>:
init.**s-**<constant expression> = *undefined* ⇒*init.virtuality0*∈{**redefined**, **finalized**}

The <constant expression> may only be omitted if <virtuality> is **redefined** or **finalized**.

Transformations

<variable definition>(*undefined*, < <variables of sort>(
<variables of sort gen name>(*name*,*) >, *sort*, *undefined*) >)
provided
getEntityDefinition0(*sort*, **sort**). *getDefaultInitialization* ≠ *undefined*
=8=>
<variable definition>(*undefined*, < <variables of sort>(
<variables of sort gen name>(*name*,*) >, *sort*,
getEntityDefinition0(*sort*, **sort**). *getDefaultInitialization*) >)

A default initialization is shorthand for specifying an explicit initialization for all those variables that are declared to be of <sort>, but where the <variable definition> was not given a <constant expression>.

If no <default initialization> is given in <syntype definition>, then the syntype has the <default initialization> of the <parent sort identifier> provided its result is in the range.

Any sort that is defined by an <object data type definition> is implicitly given a <default initialization> of Null, unless an explicit <default initialization> was present in the <object data type definition>.

Any pid sort is treated as if implicitly given a <default initialization> of Null.

If the <constant expression> is omitted in a redefined default initialization, the explicit initialization is not added.

Auxiliary functions

The function *getDefaultInitialization* computes the default initialization for a data type or syntype.

```
getDefaultInitialization(type: <data type definition> ∪ <syntype definition>): BOOLEAN =def
    case type in
    | <syntype definition>
            (*, <syntype definition syntype>(*, init = <default initialization>(*,*), *)
        => init
    | <syntype definition>(*, <syntype definition syntype>(parent, *, *)
        => parent.getDefaultInitialization
    else
    let init = take({d ∈ <default initialization>: isDefinedIn0(d, type)}) in
        if init = undefined then
            let parentSort = type.derivedDataType0 in
                if parentSort = undefined then
                    if type.isPidSort0 then
                        <operator application>(<operation identifier>(empty, "Null"), empty)
```

```
            else
                undefined
            endif
        else
            getDefaultInitialization(parentsort)
        endif
    endlet
else
    init
endif
```

F2.2.9.9.14 Now expression

Abstract syntax

Now-expression :: ()

Concrete syntax

<now expression> :: ()

Mapping to abstract syntax

| <now expression> => **mk-***Now-expression*()

F2.2.9.9.15 Import expression

Concrete syntax

<import expression> :: <remote variable><identifier> <communication constraints>

Transformations

Further study is needed for the transformations of <import expression>.

The import expression has implied syntax for the importing of the result as defined in clause 10.6 of [ITU-T Z.102] and it also has an implied Variable-access of the implied variable for the import in the context where the <import expression> appears.

The use of <import expression> in an expression is shorthand for inserting a task just before the action, where the expression occurs which assigns an implicit variable the result of the <import expression> and then uses that implicit variable in the expression. If <import expression> occurs several times in an expression, one variable is used for each occurrence.

F2.2.9.9.16 Pid expression

Abstract syntax

Pid-expression = *Self-expression*
| *Parent-expression*
| *Offspring-expression*
| *Sender-expression*

Self-expression :: ()

Parent-expression :: ()

Offspring-expression :: ()

Sender-expression :: ()

Concrete syntax

```
<pid expression> =
        <self expression>
    |   <parent expression>
    |   <offspring expression>
    |   <sender expression>
```

<self expression> :: ()

<parent expression> :: ()

<offspring expression> :: ()

<sender expression> :: ()

<create expression> = <create body>

Transformations

The use of <create expression> in an expression is a shorthand for inserting a create request just before the action where the <create expression> occurs followed by an assignment of offspring to an implicitly declared anonymous variable of the same sort as the static sort of the <create expression>. The implicit variable is then used in the expression. If <create expression> occurs several times in an expression, one distinct variable is used for each occurrence. In this case the order of the inserted create requests and variable assignments is the same as the order of the <create expression>s.

If the <create expression> contains an <agent type identifier> then the transformations that are applied to a create statement that contains an <agent type identifier> are also applied to the implicit create statements resulting from the transformation of a <create expression> (see clause 11.13.2 of [ITU-T Z.103]).

Mapping to abstract syntax

| <self expression> => **mk-***Self-expression*()
| <parent expression> => **mk-***Parent-expression*()
| <offspring expression> => **mk-***Offspring-expression*()
| <sender expression> => **mk-***Sender-expression*()

F2.2.9.9.17 Timer active expression and timer remaining duration

Abstract syntax

Timer-active-expression :: *Timer-identifier Expression**

Timer-remaining-duration :: *Timer-identifier Expression**

Further study is required to give the static formal semantics for *Timer-remaining-duration*.

Conditions on abstract syntax

∀*t* ∈ *Timer-active-expression* ∪ *Timer-remaining-duration*:
let *d* = *getEntityDefinition1*(*t*.**s**-*Timer-identifier*, **timer**) **in**
t.**s**-*Expression*.*length* = *d*.**s**-*Sort-reference-identifier*. *length* ∧
(∀*i* ∈ 1.. *t*.**s**-*Expression*. *length*:
isCompatibleTo1(*t*.**s**-*Expression*[*i*].*staticSort1*, *d*.**s**-*Sort-reference-identifier*[*i*]))
endlet

The sorts of the *Expression* list in the *Timer-active-expression* or *Timer-remaining-duration* must correspond by position to the *Sort-reference-identifier* list directly following the *Timer-name* identified by the *Timer-identifier*.

Concrete syntax

<timer active expression> :: <timer<identifier> [<expression list>]

<timer remaining duration> :: <timer<identifier> [<expression list>]

Mapping to abstract syntax

| <timer active expression>(*id*,*l*) =>
mk-*Timer-active-expression*(*Mapping*(*id*), *Mapping*(*l*))

| <timer remaining duration>(*id*,*l*) =>
mk-*Timer-remaining-duration*(*Mapping*(*id*), *Mapping*(*l*))

F2.2.9.9.18 Active agents expression

Abstract syntax

Active-agents-expression :: { *Agent-identifier* | **THIS** }

Concrete syntax

<active agents expression> :: { <agent<identifier> | **this** }

Conditions on abstract syntax

∀*exp*∈<create body>: (*exp*.**s-implicit** = **this**) ⇒
(*exp.surroundingScopeUnit0*∈<agent type definition>) ∧
(*exp.surroundingScopeUnit0.surroundingScopeUnit0*∈<agent type definition>)

this shall only be specified in an <agent type diagram> and in scopes enclosed by an <agent type diagram> and represents **THIS**.

Mapping to abstract syntax

Further study needed.

F2.2.9.9.19 Any expression

Abstract syntax

Any-expression :: *Sort-reference-identifier*

Concrete syntax

<any expression> :: <sort>

Conditions on abstract syntax

∀*exp*∈<any expression>: *isContainingElements0*(*exp*.**s-**<sort>)

The <sort> must contain elements.

Mapping to abstract syntax

| <any expression>() => **mk-***Any-expression*

Auxiliary functions

isContainingElements0(*s*:<sort>):*BOOLEAN*=$_{\text{def}}$
let *d* = *getEntityDefinition0*(*s*, **sort**) **in**
(*d*∈*PREDEFINEDSORT0*)∨
(∃*cons*∈<data type constructor>: *cons.surroundingScopeUnit0* = *d* ∧
(*cons*∈<structure definition>∪<choice definition>⇒
∀*sort*∈<sort>:*sort* **in** *cons.fieldSortList0*⇒*isContainingElements0*(*sort*)))∨
(*d.specialization0*≠*undefined*∧
isContainingElements0(*d.specialization0*.**s-**<type expression>.*baseType0*)∧
(∀*acp*∈<actual context parameter>:
acp **in**
d.actualContextParameterList0∧*acp.idKind0* = **sort**⇒ *isContainingElements0*(*acp*))
endlet

F2.2.9.9.20 State expression

Abstract syntax

State-expression :: ()

Concrete syntax

<state expression> :: ()

Mapping to abstract syntax

| <state expression>() => **mk-***State-expression*

F2.2.9.9.21 Value returning procedure call

Abstract syntax

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

Concrete syntax

<value returning procedure call> =
<procedure call body>
| <remote procedure call body>

Conditions on concrete syntax

∀*exp*∈<continuous expression>: *exp.parentAS0*∈<continuous signal> ⇒
¬∃*procCall*∈<value returning procedure call>: *isAncestorAS0*(*exp*, *procCall*)

∀*exp*∈<provided expression>: *exp*. *parentAS0*∈<input part> ⇒
¬∃*procCall*∈<value returning procedure call>: *isAncestorAS0* (*exp*, *procCall*)

A <value returning procedure call> must not occur in the <Boolean expression> of a <continuous signal> or <enabling condition>.

∀*procId*∈<identifier>: *procId*. *parentAS0*∈<value returning procedure call>⇒
getEntityDefinition0(*procId*, **procedure**). **s-**<procedure heading>.**s-**<procedure result> ≠ *undefined*

The <procedure identifier> in a <value returning procedure call> must identify a procedure having a <procedure result>.

∀*procCall*∈<procedure call body>:
procCall ∈<value returning procedure call> ∧ *procCall*.**s-this** ≠ *undefined* ⇒
getEntityDefinition0(*procId*, **procedure**) =
parentAS0ofKind(*procCall*, <procedure definition>)

If **this** is used, <procedure identifier> must denote an enclosing procedure.

Transformations

let *nn*= *newName* **in**
p=<value returning procedure call>(<procedure call body>(*id*, *params*))
provided *parentAS0ofKind*(*id.refersto0*, <agent type definition>) ≠
parentAS0ofKind(*p*, <agent type definition>)
=8=>
let *par*=*parentAS0ofKind*(*p*, <agent type definition>) **in**
<value returning procedure call>(<procedure call body>(
<identifier>(*par.fullQualifier0*⌒<path item>(*par.entityKind0*, *par.entityName0*), *nn*),
params))
endlet
and // add the new definition
let *defs*=
parentAS0ofKind(*p*, <agent type definition>).**s-**<agent structure>.**s-**<entity in agent>**-seq in**
defs => *defs* ⌒
<internal procedure definition>(*empty*,
<procedure heading>(
<procedure preamble>(*undefined*, *undefined*),
empty, *nn*, *empty*, *undefined*, *undefined*, *empty*, *undefined*, *empty*),
empty,
<procedure body>(*undefined*, *undefined*, *empty*))
endlet

If the <procedure identifier> is not defined within the enclosing agent, the procedure call is transformed into a call of a local, implicitly created, subtype of the procedure.

The keyword **this** implies that when the procedure is specialized, the <procedure identifier> is replaced by the identifier of the specialized procedure.

```
let nn= newName in
p=<value returning procedure call>(<remote procedure call body>(id, params, constrs, onexc)
=>
<value returning procedure call>(<procedure call body>(nn, empty))
and // add the new definition
    let defs=
        parentAS0ofKind(p, <agent type definition>).s-<agent structure>.s-<entity in agent>-seq in
        defs => defs ⌒
            <internal procedure definition>(empty,
                <procedure heading>(
                    <procedure preamble>(undefined, undefined),
                    empty, nn, empty, undefined, undefined, empty, undefined, empty),
                empty,
                <procedure body>(undefined,
                    <start>(undefined, undefined, undefined,
                        <terminator>(undefined,
                            <return>(<return body>(p)))), empty))
    endlet
```

When the <value returning procedure call> contains a <remote procedure call body>, the following procedure with an anonymous name referred to as RPCcall is implicitly defined. RPCsort is the <sort> in <procedure result> of the procedure definition denoted by the <procedure identifier>.

```
procedure RPCcall -> RPCsort;
    start;
    return call <remote procedure call body>;
endprocedure;
```

NOTE – This transformation is not again applied to the implicit procedure definition.

Mapping to abstract syntax

```
| <value returning procedure call>(<procedure call body>(t, id, params)) =>
    mk-Value-returning-call-node(Mapping(t), Mapping(id), Mapping(params))
```

F2.2.9.9.22 Type coercion

Abstract syntax

Type-coercion :: *Expression Sort-reference-identifier*

Concrete syntax

<type coercion> :: <expression> <sort>

Conditions on concrete syntax

Further study needed.

Transformations

Further study needed.

Mapping to abstract syntax

Further study needed.

F2.3 Transformation of SDL-2010 shorthands

This clause details the transformation of the SDL-2010 constructs, whose dynamic semantics are given after a transformation to the subset of SDL-2010 for which *Abstract Grammar* exists. These shorthand notations are constructs for which a *Model* section exists.

The properties of a shorthand notation are derived from the way it is modelled in terms of (or transformed to) the primitive concepts. In order to ensure easy and unambiguous use of the shorthand notations, and to reduce side effects when several shorthand notations are combined, these concepts are transformed in a specified order as detailed in this clause.

The specified order of transformation means that in the transformation of a shorthand notation of order *n*, another shorthand notation of order *m* may be used, provided m>n. The order of the transformation is given as a number inside the transformation arrow, e.g., =5=> for a transformation of order 5.

The transformations are described as a number of enumerated steps. One step may describe the transformation of several concepts and thus consist of a number of sub-steps, either because these concepts must be transformed as a group or because the transformation order between these concepts is not significant.

If entities are moved to different scopes during the subsequent transformation steps, the <qualifier>s in every <identifier> bound to such an entity are updated to reflect this change. In fact, this case should not happen in the new version of SDL-2010.

The following enumeration details the transformation steps to be performed in order.

1) Lexical transformations:
   a) <macro definition> items and <macro call> items (see clause 6.7 of [ITU-T Z.102]) are identified lexically and <macro call> items are expanded;
   b) <macro definition> items are removed (also in <package definition> items).
   – These transformations are not described formally, i.e., no macros are considered in the formal semantics.
   – This step also includes simple transformations that just adapt the AS0.
2) <operation definition> items are transformed into procedures having anonymous names and having the result as <procedure result>. See clause 12.1.7 of [ITU-T Z.101] and clause 12.1.7 of [ITU-T Z.104].
3) <task> items, <task area> items, and <statements> lists are transformed as defined in clause 11.13.1 of [ITU-T Z.103], clause 11.14 of [ITU-T Z.102] and clause 11.14 of [ITU-T Z.103].
4) Definition references are replaced by <referenced definition>s (see clause 7.3 of [ITU-T Z.101]).
5) The graphs are normalized:
   – non-terminating decisions and non-terminating transition options are transformed into terminating decisions and terminating transition options respectively;
   – the actions and/or terminator statement following the decisions and transition options are moved to appear as <free action>s. Those generated <free action>s which have no label attached are given anonymous labels;
   – action lists (including the terminator statement which follows) where the first action (if any, otherwise the following terminator statement) has a label attached, are replaced by a join to the label and the action list appears as a <free action>.
6) The package Predefined is included in the <sdl specification>.
7) Transformation of generic system (see clause 13 of [ITU-T Z.103]) and external data (an external synonym definition item – see clause 12.1.8.3 of [ITU-T Z.104], an operation

defined by an external operation definition – see clause 12.1.7 of [ITU-T Z.104], a procedure defined by an external procedure definition – see clause 9.4 of [ITU-T Z.103], or <informal text> in a transition option):

- identifiers in <simple expression>s contained in the <sdl specification> are bound to definitions. During this binding, only <data definition>s defined in the predefined package Predefined and <external synonym definition>s are considered (that is, all other <data definition>s are ignored);
- <external synonym> items are replaced by <synonym definition>s and informal text in transition options is replaced by <range condition>. How this is done is not defined by SDL-2010;
- <simple expression> items are evaluated and <select definition>s, <option area>s, <transition option>s and <transition option area>s are removed.

8) Transformation of:

- Non-deterministic decision (see clause 11.13.5 of [ITU-T Z.102]);
- Operations involving <infix operation name>s and their operands transformed to the prefix form (see clause 12.2.1 of [ITU-T Z.104]);
- Structure data type (<structure definition>: see clause 12.1.6.2 of [ITU-T Z.101] and clause 12.1.6.2 of [ITU-T Z.104]);
- State list (<state list>: see clause 11.2 of [ITU-T Z.103]);
- More than one <stimulus> or asterisk in an <input list> (see clause 11.3 of [ITU-T Z.103]);
- More than one <save item> or asterisk in a <save list> (see clause 11.7 of [ITU-T Z.103]);
- Field primary (see clause 12.2.3 of [ITU-T Z.101] and clause 12.2.3 of [ITU-T Z.104]);
- Composite primary (for structure or array values: or clause 12.2.3 of [ITU-T Z.101]);
- The range constraint for <syntype definition>s (see clause 12.1.8.2 of [ITU-T Z.101]);
- Multiple signals in <output body> (see clause 11.13.4 of [ITU-T Z.103]);
- Multiple timers in <set body> and <reset body> (see clause 11.15 of [ITU-T Z.103]);
- Channel to channel connections replacing them with gates on the (implicit) agent type (see clause 10.2 of [ITU-T Z.103]);
- Default duration value for timer set (see clause 11.15 of [ITU-T Z.101]);
- Initialization of variables of sorts with default initialization (see clause 12.3.1 of [ITU-T Z.101], clause 12.3.3.2 of [ITU-T Z.101] and clause 12.3.3.2 of [ITU-T Z.104]);
- <stimulus> containing <indexed variable>s and <field variable>s are transformed (see clause 11.3 of [ITU-T Z.103]);
- Replacing interface identifiers from interface or signallist definitions to a list of signal identifiers (see clause 10.4 Semantics of [ITU-T Z.101]);
- Indexed primary (see clause 12.2.3 of [ITU-T Z.101] and clause 12.2.3 of [ITU-T Z.104]);
- Field variable (see clause 12.3.3.1 of [ITU-T Z.101]);
- Indexed variable (see clause 12.3.3.1 of [ITU-T Z.101]);
- <return body> with <expression> (see clause 11.12.2.4 of [ITU-T Z.101]).

9) Insertion of implicit channels as described in clause 10.1 of [ITU-T Z.103].

10) Insertion of implicit signal lists as described in clauses 10.5 and 10.6 of [ITU-T Z.102].

11) Replacement of context parameters described in clause 8.3 of [ITU-T Z.103].

12) Full qualifiers are inserted:

- According to the visibility rules and the rules for resolution by context (see clause 6.6 of [ITU-T Z.101]), qualifiers are extended to denote the full path.

13) Transformation of asterisk state:

- A body originating from an agent definition or procedure definition has its asterisk states expanded according to the model defined in clause 11.2 of [ITU-T Z.103].
- Multiple appearance of state is merged (see clause 11.2 of [ITU-T Z.103]).

14) Implicit declarations for remote procedures and remote variables (see clauses 10.5 and 10.6 of [ITU-T Z.102]) are generated. Imported and exported values (see clause 10.6 of [ITU-T Z.102])) are transformed. Then remote procedures (see clause 10.5 of [ITU-T Z.102])) are transformed.

SERIES OF ITU-T RECOMMENDATIONS

Series A	Organization of the work of ITU-T
Series D	Tariff and accounting principles and international telecommunication/ICT economic and policy issues
Series E	Overall network operation, telephone service, service operation and human factors
Series F	Non-telephone telecommunication services
Series G	Transmission systems and media, digital systems and networks
Series H	Audiovisual and multimedia systems
Series I	Integrated services digital network
Series J	Cable networks and transmission of television, sound programme and other multimedia signals
Series K	Protection against interference
Series L	Environment and ICTs, climate change, e-waste, energy efficiency; construction, installation and protection of cables and other elements of outside plant
Series M	Telecommunication management, including TMN and network maintenance
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Telephone transmission quality, telephone installations, local line networks
Series Q	Switching and signalling, and associated measurements and tests
Series R	Telegraph transmission
Series S	Telegraph services terminal equipment
Series T	Terminals for telematic services
Series U	Telegraph switching
Series V	Data communication over the telephone network
Series X	Data networks, open system communications and security
Series Y	Global information infrastructure, Internet protocol aspects, next-generation networks, Internet of Things and smart cities
Series Z	**Languages and general software aspects for telecommunication systems**

Printed in Switzerland
Geneva, 2017