INTERNATIONAL TELECOMMUNICATION UNION

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Z.100
Annex F2
(11/2000)

SERIES Z: LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

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

Specification and description language (SDL)

Annex F2: SDL formal definition: Static semantics

ITU-T Recommendation Z.100 – Annex F2

(Formerly CCITT Recommendation)

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	Z.120–Z.129
PROGRAMMING LANGUAGES	
CHILL: The ITU-T programming 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.399
QUALITY OF TELECOMMUNICATION SOFTWARE	Z.400–Z.499
METHODS FOR VALIDATION AND TESTING	Z.500–Z.599

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

ITU-T Recommendation Z.100

Specification and description language (SDL)

ANNEX F2
SDL formal definition: Static semantics

Summary

This Annex F2 describes the static semantic constraints, and transformations as identified by the Model clauses of ITU-T Z.100.

Source

Annex F2 to ITU-T Recommendation Z.100 was prepared by ITU-T Study Group 10 (2001-2004) and approved under the WTSA Resolution 1 procedure on 24 November 2000.

FOREWORD

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

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

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

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

NOTE

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

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, implementors are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database.

© ITU 2002

All rights reserved. No part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from ITU.

CONTENTS

Page

1 Overview of the Static Semantics 1
1.1 Grammar 1
1.2 Well-formedness Conditions 1
1.3 Transformation Rules 2
1.4 Definitions Used from Annex F1 2
2 Static Semantics 3
2.1 General Definitions 3
2.1.1 Division of Text 3
2.1.2 Abstraction of the Concrete Grammar (AS0) 4
2.1.3 Static Conditions 4
2.1.4 Transformation Rules 4
2.1.5 Mapping Rules 5
2.1.6 Predefinition 5
2.2 Visibility Rules, Names and Identifiers 13
2.2.1 Name 13
2.2.2 Identifier 13
2.2.3 Path Item 32
2.3 Informal Text 33
2.4 General Framework 33
2.4.1 SDL Specification 33
2.4.2 Package 33
2.4.3 Referenced definition 36
2.4.4 Select Definition 39
2.4.5 Transition Option 40
2.4.6 Associations 40
2.5 Structural Concepts 41
2.5.1 Structural Type Definitions 41
2.5.2 Type Expression 45
2.5.3 Definitions Based on Types 48
2.5.4 Abstract Type 50
2.5.5 Type Reference 50
2.5.6 Gate 50
2.5.7 Context parameters 54
2.5.8 Specialization 65

Page

2.6 Agents ... 72
2.6.1 System ... 75
2.6.2 Block ... 76
2.6.3 Process ... 76
2.6.4 Procedure ... 76
2.7 Communication ... 81
2.7.1 Channel Definition ... 81
2.7.2 Connections ... 86
2.7.3 Signal ... 87
2.7.4 Signal List Definition ... 88
2.7.5 Remote Procedures ... 89
2.7.6 Remote Variables ... 96
2.8 Behaviour ... 100
2.8.1 Start ... 100
2.8.2 State ... 101
2.8.3 Input ... 103
2.8.4 Priority Input ... 105
2.8.5 Continuous Signal ... 105
2.8.6 Enabling Condition ... 105
2.8.7 Save ... 106
2.8.8 Implicit Transition ... 107
2.8.9 Spontaneous Transition ... 107
2.8.10 Label ... 107
2.8.11 State Machine and Composite State ... 108
2.8.12 Transition ... 115
2.8.13 Action ... 121
2.8.14 Statement List ... 129
2.8.15 Timer ... 136
2.8.16 Exception ... 138
2.9 Data ... 142
2.9.1 Data Definitions ... 142
2.9.2 Data Type Definition ... 142
2.9.3 Interface Type ... 145
2.9.4 Specialization of Data Types ... 147
2.9.5 Operations ... 150
2.9.6 Any ... 152

Page

2.9.7 Data Type Constructors 153
2.9.8 Behaviour of Operations 162
2.9.9 Additional Data Definition Constructs 166
2.9.10 Expression 173
3 Transformation of SDL Shorthands 190
APPENDIX I 192
INDEX 192
I.1 Functions 192
I.2 Domains 197
I.3 AS0 Nonterminals 197
I.4 AS1 Nonterminals 205

ITU-T Recommendation Z.100

Specification and description language (SDL)

ANNEX F2
SDL formal definition: Static semantics

1 Overview of the Static Semantics

In order to define the formal semantics of SDL, the language definition is decomposed into several parts:

- grammar;
- well-formedness;
- transformation rules; and
- dynamic semantics.

The starting point for defining the formal semantics of SDL is therefore a syntactically correct SDL specification, represented as an abstract syntax tree (AST).

The first three parts of the formal semantics are collectively referred to as static semantics in the context of SDL.

The *grammar* defines the set of syntactically correct SDL specifications. In ITU-T Z.100, a concrete textual, a concrete graphical, and an abstract grammar are defined formally using the Backus-Naur Form (BNF) with some extensions for capturing the graphical part. The abstract grammar is obtained from the concrete grammars by removing irrelevant details such as separators and lexical rules.

The *well-formedness conditions* define which specifications, that are correct with respect to the grammar, are also correct with respect to context information, such as which names it is allowed to use at a given place, which kind of values it is allowed to assign to variables etc.

Furthermore, some language constructs appearing in the concrete grammars are replaced by other language elements in the abstract grammar using *transformation rules* to keep the set of semantic core concepts small. These transformations are described in the model paragraphs of ITU-T Z.100. Formally they are represented as rewrite rules.

The *dynamic semantics* is given only to syntactically correct SDL specifications that satisfy the well-formedness conditions. The dynamic semantics defines the set of computations associated with a specification. The dynamic semantics is formalised within Annex F3, where also an example for the definition of the semantics is presented.

1.1 Grammar

Formalising a grammar is a well-known task. As usual, this is done in SDL using a grammar representation in BNF (Backus-Naur Form). However, the grammar in ITU-T Z.100 is designed to be a presentation grammar, i.e. it is not made to generate a parser automatically. Moreover, some restrictions that finally guarantee uniqueness of the semantics could not be formulated formally in BNF and have been stated in the text instead. Therefore, the grammar is defined using BNF and some text (mostly for the precedence rules). The translation step from the concrete textual SDL syntax to AS0 is not formally defined, but is derived from the correspondence between the two syntaxes, which is almost one-to-one.

1.2 Well-formedness Conditions

There are some static conditions that must be satisfied by a well-formed SDL specification and that can be checked without execution of an instance (clause 2). An SDL specification is valid if and only if it satisfies the syntactic rules and the static conditions. In fact, the well-formedness conditions refer to the grammar, but for conciseness reasons they have not been stated in the grammar because they are not expressible in a context-free grammar.

There are basically five kinds of well-formedness conditions:

- Scope/visibility rules: the definition of an entity introduces an identifier that may be used as the reference to the entity. Only visible identifiers must be used. The scope/visibility rules are applied to determine whether the corresponding definition of an identifier is visible or not.

- Disambiguation rules: Sometimes a name might refer to several identifiers. Rules are applied to find out the correct one.
- Data type consistency rules: These rules ensure that at interpretation time, no operation is applied to operands that do not match their argument types. More specifically, the data type of an actual parameter must be compatible with that of the corresponding formal parameter; the data type of an expression must be compatible with that of the variable to which the expression is assigned.
- Special rules: There are some rules applicable to specific entities. For example, there must exist at least one block definition or a process definition in a system definition.
- Plain syntax rules: There are some rules that refer to the correctness of the plain syntax and that do not get transformed into the abstract syntax, e.g. that the name at the beginning and at the end of a definition match.

The static semantics of SDL is defined in terms of first order predicate calculus.

1.3 Transformation Rules

For a language with a rich syntax it is essential to define the semantics only for the core concepts and transform the other constructs into this core. In Figure F2-1, the general approach is shown. The language is defined with its concrete grammar using lexical and syntax rules. Consistency constraints are defined on this concrete grammar.

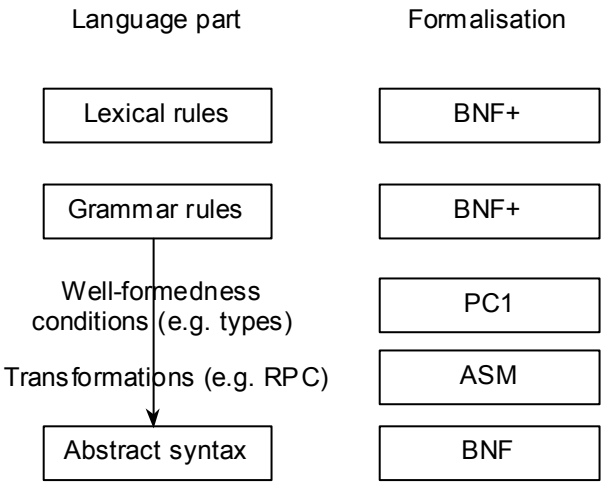


Figure F2-1/Z.100: Formalisms to Define the Static Aspects of SDL

However, the meaning is given only to the core concepts that are within the Abstract Syntax. In order to be able to do so, a transformation is given in clause 2, mapping the concrete grammar to the abstract grammar. It is important here to correctly identify the core concepts in order to facilitate the transformation. If there are too many concepts, giving a semantics is unnecessarily complicated. If there are too few or the wrong concepts, the transformations tend to be very complex, and their meaning is no longer easily understood. In the scope of SDL, object-orientation was introduced in SDL-92, but still the (formal) semantics definition and the abstract syntax relied on instances and had no notion of class. The result was a very cumbersome transformation. This is rectified in the formal semantics of SDL-2000.

With a proper and small abstract syntax it is possible to define a formal semantics of the language. However, in order to have a valid formal semantics, it is also necessary to define the transformations formally. This is achieved using a suitable formalisation for the transformations. In the case of SDL, rewrite rules were chosen to define the transformations. These rules define patterns of the AST which are to be replaced by other AST patterns. In fact, several groups of such rewrite rules are defined that are applied in turn.

1.4 Definitions Used from Annex F1

The following definitions for the syntax and semantics of abstract state machines (ASM) are used within this annex. They are defined in Annex F1. They are introduced here for cross-referencing reasons.

– the keywords **domain**, **static**, **initially**, **controlled**, **monitored**, **shared**, **constraint**;

– the domains *AGENT*, *PROGRAM*, *X*, *BOOLEAN*, *NAT*, *REAL*, *TOKEN*, *DefinitionAS1*, *DefinitionAS0*;

– the functions *take*, *currentTime*, *program*, *Self*, *undefined*, *True*, *False*, *empty*, *head*, *tail*, *last*, *length*, *toSet*, *parentAS1*, *parentAS0*, *parentAS0ofKind*, *parentAS1ofKind*, *isAncestorAS0*, *isAncestorAS1*, *rootNodeAS1*, *rootNodeAS0*, *replaceInSyntaxTree*;

– the operation symbols *****, +, **-set**, =, ≠, ∧, ∨, ⇒, ⇔, ¬, ∃, ∀, >, ≥, <, ≤, +, -, *****, /, **in**, ×, ⁀, ∪, ∩, \, ∈, ∉, ⊆, ⊂, | |, ⋃, ∅, **mk-**, **s-**.

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

2 Static Semantics

In this clause, the static semantics of SDL is formalised. There are essentially three parts to be defined, namely the transformations, the mapping and the static well-formedness rules. The transformations describe replacements within the AS0, whereas the mapping describes how AS0 productions are mapped to AS1 productions. For the well-formedness rules there are two areas, namely rules for the AS1 and rules for the AS0. All the rules are defined in terms of first order predicate calculus. Predicate calculus is a well-known technique that can be used for formal definition. The definition of static semantics is independent of implementation.

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

The context conditions 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 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.

As an 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:

$$\forall d, d' \in \mathit{ENTITYDEFINITION}_1 : d.\mathit{entityKind}_1 = d'.\mathit{entityKind}_1 \wedge d \neq d' \Rightarrow d.\mathit{identifier}_1 \neq d'.\mathit{identifier}_1$$

where:

d and d' represent any two abstract syntax tree nodes belonging to the set $\mathit{ENTITYDEFINITION}_1$.

$d.\mathit{entityKind}_1$ and $d'.\mathit{entityKind}_1$ get the entity kinds of the corresponding syntax constructs.

$d.\mathit{identifier}_1$ and $d'.\mathit{identifier}_1$ get the full identifiers of the corresponding syntax constructs.

2.1 General Definitions

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 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 clause 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.

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 textual grammar (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 simply parsing algorithm from the concrete grammars PR and GR.

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

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.

2.1.4 Transformation Rules

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

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

<dependent transformation>::=

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

The pattern as well as the expression refer to the syntax as defined for ASM in Annex F1. 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 3.

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.

2.1.5 Mapping Rules

The mapping rules in fact 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** clauses. This is preceded with the following header part and followed by an **endcase**.

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

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

2.1.6 Predefinition

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

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** $=_{def}$
if *s* = *empty* **then** *empty* **else** *s*. *head* ⌢ *BigSeq*(*s.tail*) **endif**

2.1.6.2 Domain Definitions for AS1

SCOPEUNIT$_1$: the union of all the scope unit in AS1.

SCOPEUNIT$_1$ $=_{def}$ *Package-definition*
- ∪ *Agent-definition*
- ∪ *Agent-type-definition*
- ∪ *Procedure-definition*
- ∪ *Signal-definition*
- ∪ *Composite-state-type-definition*
- ∪ *Data-type-definition*
- ∪ *State-node*

ENTITYDEFINITION$_1$: the union of all the entity definitions in AS1.

ENTITYDEFINITION$_1$ $=_{def}$ *Package-definition*
- ∪ *Agent-definition*
- ∪ *Agent-type-definition*
- ∪ *Procedure-definition*
- ∪ *Composite-state-type-definition*
- ∪ *Channel-definition*
- ∪ *Gate-definition*
- ∪ *Signal-definition*
- ∪ *Timer-definition*
- ∪ *Exception-definition*
- ∪ *Variable-definition*
- ∪ *Data-type-definition*
- ∪ *State-node*

∪ *Syntype-definition*
∪ *Literal-signature*
∪ *Operation-signature*

ENTITYKIND$_1$: the set of all the entity kinds in AS1.

ENTITYKIND$_1$ =$_{def}$ { **agent**, **agent type**, **package**, **state**, **state type**, **procedure**, **variable**, **signal**, **timer**, **channel**, **gate**, **sort**, **exception**, **literal**, **operation**}

AGENTKIND$_1$: the set of agent kinds in AS1.

AGENTKIND$_1$ =$_{def}$ {**system**, **block**, **process**}

SIGNAL$_1$=$_{def}$ {*id*∈ *Identifier*: *id.idKind*$_1$∈{**signal**, **timer**}}

TYPEDEFINITION$_1$=$_{def}$ *Agent-type-definition*∪*Procedure-definition*∪*Composite-state-type-definition*∪ *Data-type-definition*

VALUE$_1$=$_{def}$ *Literal-signature*
Domain Definitions for AS0

ENTITYKIND$_0$ =$_{def}$ {**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**}

TYPEDEFINITION$_0$ =$_{def}$ <agent type definition> ∪ <composite state type definition> ∪
<data type definition> ∪ <procedure definition> ∪ <interface definition> ∪ <signal definition item>

FORMALCONTEXTPARAMETER$_0$ =$_{def}$ <agent type context parameter> ∪
<agent context parameter> ∪ <procedure context parameter> ∪ <remote procedure context parameter> ∪
<signal context parameter gen name> ∪ <variable context parameter gen name> ∪
<remote variable context parameter gen name> ∪ <timer context parameter gen name> ∪
<synonym context parameter gen name> ∪ <sort context parameter> ∪
<exception context parameter gen name> ∪ <composite state type context parameter> ∪
<gate context parameter> ∪ <interface context parameter gen name>

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

PREDEFINEDDEFINITION$_0$=$_{def}$*PREDEFINEDLITERAL*$_0$ ∪*PREDEFINEDOPERATION*$_0$∪*PREDEFINEDSORT*$_0$

PREDEFINEDLITERAL$_0$, ,*PREDEFINEDOPERATION*$_0$ and *PREDEFINEDSORT*$_0$ represent all the literals, operations and sorts defined within the package Predefined seperately.

ENTITYDEFINITION$_0$ =$_{def}$ <package definition> ∪ <agent definition> ∪ <composite state> ∪
<textual typebased agent definition> ∪<synonym definition> ∪<channel definition>
<textual typebased state partition definition> ∪ <syntype definition> ∪ <exception definition item> ∪
<timer definition item> ∪ <signal list definition> ∪ <parameters of sort> ∪ <variables of sort> ∪
<literal signature> ∪ <operation signature> ∪ <textual gate definition> ∪
<remote variable definition gen name> ∪ *TYPEDEFINITION*$_0$ ∪ *FORMALCONTEXTPARAMETER*$_0$∪
PREDEFINEDDEFINITION$_0$

TYPEREFERENCE$_0$ =$_{def}$ <agent type reference> ∪ <composite state type reference> ∪
<data type reference> ∪ <procedure reference> ∪ <signal reference> ∪ <interface reference>

REFERENCE$_0$ =$_{def}$ <package reference> ∪ <agent reference> ∪ <composite state reference> ∪
<textual operation reference> ∪*TYPEREFERENCE*$_0$

SCOPEUNIT$_0$ =$_{def}$ <package definition> ∪ <agent definition> ∪ <operation definition> ∪ <composite state> ∪
<sort context parameter> ∪ <signal context parameter gen name> ∪ <compound statement>
∪*TYPEDEFINITION*$_0$

SIGNAL$_0$=$_{def}$ {*id*∈ <identifier>: *id.idKind*$_0$∈ {**signal**, **timer**, **remote procedure**, **remote variable**}}

SYMBOL$_0$=$_{def}$ {"<",">","<=",">=","Length"}

CONTEXT$_0$=$_{def}$ <assignment> ∪ <decision> ∪ <expression>

BINDING$_0$ =$_{def}$ <name> × *ENTITYDEFINITION*$_0$

BINDINGLIST$_0$ =$_{def}$ *BINDING*$_0$*

2.1.6.3 Function Definitions on AS1

Auxiliary functions on the AS1

The function *identifier*$_1$ is used to get the identifier with full qualifier for an entity definition in AS1.

identifier$_1$(*d*: *ENTITYDEFINITION*$_1$):*Identifier* =$_{def}$
mk-*Identifier*(*d*. *fullQualifier*$_1$, *d.entityName*$_1$)

The function *fullQualifier*$_1$ is used to get the full qualifier for an entity definition in AS1.

fullQualifier$_1$(*d*: *ENTITYDEFINITION*$_1$): *Qualifier* =$_{def}$
let *su*= *parentAS1ofKind*(*d*, *SCOPEUNIT*$_1$) **in**
if *su* = *undefined* **then** *empty*
elseif *d.entityKind*$_1$∈ {**operation**, **literal**}∧*su* ∈ *Data-type-definition* **then** *su.fullQualifier*$_1$
else *su.fullQualifier*$_1$^**mk**-*Qualifier*(*su.entityKind*$_1$, *su.entityName*$_1$)
endif
endlet

idKind$_1$(*id*:*Identifier*):*ENTITYKIND*$_1$=$_{def}$
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*∪
Exception-definition∪*Formal-argument*∪*Variable-definition*∪*Any-expression*: **sort**
| *Set-node*∪*Reset-node*∪*Timer-active-expression*: **timer**
| *Originating-gate*∪*Destination-gate*: **gate**
| *Raise-node*: **exception**
| *Composite-state-type-definition*∪*State-machine-definition*∪*State-node*∪*State-partition*: **state type**
| *Literal* : **literal**
| *Open-range*∪*Operation-application*: **opration**
| *Input-node*:
if *id* **in** *id.parentAS1*.**s-***Variable-identifier*-**seq then variable**
else signal
endif
| *Handle-node*:
if *id* **in** *id.parentAS1*.**s-***Variable-identifier*-**seq then variable**
else exception
endif
| *Variable-access*∪*Assignment-attempt*∪*Assignment*: **variable**
| *Direct-via*:
if *getEntityDefinition*$_1$(*id*, **channel**) ≠ *undefined* **then channel**
else gate
endif
endcase

The function *entityName*$_1$ is used to get the entity name for an entity definition in AS1.

entityName$_1$(*d*: *ENTITYDEFINITION*$_1$): *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*
| *Exception-definition* => *d*.**s-***Exception-name*
| *Variable-definition* => *d*.**s-***Variable-name*
| *Value-data-type-definition* => *d*.**s-***Sort*
| *Object-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 $entityKind_1$ is used to get the entity kind for an entity definition on AS1.

$entityKind_1$(*d*: $ENTITYDEFINITION_1$): $ENTITYKIND_1$ =$_{def}$
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**
| *Exception-definition* => **exception**
| *Variable-definition* => **variable**
| *Data-type-definition* => **sort**
| *Syntype-definition* => **sort**
| *Literal-signature* => **literal**
| *Operation-signature* => **operation**
otherwise *undefined*
endcase

The function $getEntityDefinition_1$ gets the entity definition for an identifier.

$getEntityDefinition_1$(*id*: *Identifier*, *k*: $ENTITYKIND_1$): $ENTITYDEFINITION_1$=$_{def}$
take({*d*∈ $ENTITYDEFINITION1$: *d.identifier1* = *id* ∧ *d.entityKind1* = *k*∧
(*d.entityKind1* = **operation** ⇒
isActualAndFormalParameterMatched1(*id.actualParameterListOfOpId1*,
d.formalParameterSortList1))})

The function $agentKind_1$ is used to get the agent kind for an *Agent-definition* or *Agent-type-definition*.

$agentKind_1$(*d*: *Agent-definition*∪*Agent-type-definition*):$AGENTKIND_1$=$_{def}$
if *d*∈ *Agent-type-definition* **then** *d*.**s-***Agent-kind*
else
let *td*=$getEntityDefinition_1$(*d*.**s-***Agent-type-identifier*, **agent type**) **in**
td.**s-***Agent-kind*
endlet
endif

value$_1$ returns the set of *VALUE*$_1$corresponding to a *Sort*.

value$_1$(*e*: *Constant-expression*):*VALUE*$_1$ =$_{def}$
case *e* **of**
| *Literal*(*) => *compute*(*e*)
| *Conditional-expression*(*bool*, *consequence*, *alternative*) =>
if *bool.value*$_1$.*semvalue* **then** *consequence.value*$_1$ **else** *alternative.value*$_1$ **endif**
| *Equality-expression*(*a*, *b*) => *computeEquality*(*a.value*$_1$, *b.value*$_1$)
| *Operation-application*(*proc*, *values*) => *compute*(*proc*, < *v* **in** *values*: (*v.value*$_1$) >
| *Range-check-expression*(*range*, *expr*) => *expr.value*$_1$ ∈ *range.range1*
endcase

range1 returns the set of elements that satisfy the condition.

range1 (*r*: *Range-condition*): *VALUE*$_1$-**set** =$_{def}$
{ *v*∈ *VALUE*$_1$:
∃*ci* ∈ *r*.**s-***Condition-item*:
((*ci* ∈ *Open-range*) ∧ *isInOpenRange*$_1$(*v*, *ci*)) ∨
((*ci*∈ *Closed-range*) ∧
(∃s_1, s_2∈ *ci*.**s-***Open-range* : ($s_1 \neq s_2$) ∧ *isInOpenRange*$_1$(*v*, s_1)∧ *isInOpenRange*$_1$(*v*, s_2)))}

isInOpenRange$_1$(*v*: *VALUE*$_1$, *r*: *Open-range*): *BOOLEAN* =$_{def}$
let *operator* = *r*.**s-***Operation-identifier*.**s-***Name* **in**
let *v′* = *r*.**s-***Constant-expression.value*$_1$ **in**
case *operator* **of**
| "=" => *v* = *v′*
| "≠" => *v* ≠ *v′*
| "≤" => *v* ≤ *v′*
| "≥" => *v* ≥ *v′*
| ">" => *v* > *v′*
| "<" => *v* < *v′*
endcase
endlet

The function *staticSort*$_1$ the static sort of *e*.

staticSort$_1$(*e*: *Expression*): *Sort-reference-identifier* =$_{def}$
case *e* **of**
| *Literal* => *getEntityDefinition*$_1$(*e*.**s-***Literal-identifier*, **literal**).**s-***Result*
| *Variable-identifier* => *getEntityDefinition*$_1$(*e*, **variable**).**s-***Sort-reference-identifier*
| *Equality-expression*∪*Range-check-expression*∪*Timer-active-expression* =>
mk-*Identifier*(**mk-***Qualifier*(*Name*("*Predefined*")), "*Boolean*")
| *Now-expression* => **mk-***Identifier*(**mk-***Qualifier*(*Name*("*Predefined*")), "*Time*")
| *Pid-expression* => **mk-***Identifier*(**mk-***Qualifier*(**mk-***Name*("*Predefined*")), "*Pid*")
| *Any-expression* => *e*.**s-***Sort-reference-identifier*
| *Operation-application* => *getEntityDefinition*$_1$(*e*.**s-***Operation-identifier*, **operation**).**s-***Result*
| *Value-returning-call-node* => *getEntityDefinition*$_1$(*e*.**s-***Procedure-identifier*, **procedure**).**s-***Result*
| *Conditional-expression* => *staticSort*$_1$(*e*.**s-***Consequence-expression*)
otherwise *undefined*
endcase

The predicate *isDirectSuperType*$_1$ is used to determine if the first entity definition is the direct super type of the second one.

isDirectSuperType$_1$(*d*: *ENTITYDEFINITION*$_1$, *d′*: *ENTITYDEFINITION*$_1$): *BOOLEAN*=$_{def}$
case *d′* **of**
| *TYPEDEFINITION*$_1$*Interface-definition* => *d* = *getEntityDefinition*$_1$(*d′*.**s-***Identifier*, *d′.entityKind*$_1$)
| *Interface-definition* =>
d∈ *Interface-definition* ∧
(∃*dataId*∈ *Data-type-identifier*: *dataId.parentAS1*= *d′* ∧ *d* = *getEntityDefinition*$_1$(*dataId*, **sort**))
| *Syntype-definition* =>
isDirectSuperType$_1$(*d*,*d′.derivedDataType*$_1$)

otherwise *False*
endcase

The predicate $isSuperType_1$ is used to determine if the first entity definition is the super type of the second one.

$isSuperType_1(d: ENTITYDEFINITION_1, d': ENTITYDEFINITION_1): BOOLEAN =_{def}$
$isDirectSuperType_1(d, d') \vee \exists\ d'' \in ENTITYDEFINITION_1: isSuperType_1(d, d'') \wedge isSuperType_1(d'', d')$

The function $derivedDataType_1$ is used to get the data type definition for a given *Syntype-definition*.

$derivedDataType_1(d: Syntype\text{-}definition \cup Data\text{-}type\text{-}definition): Data\text{-}type\text{-}definition =_{def}$
if $d \in Data\text{-}type\text{-}definition$ **then** d
else $getEntityDefinition_1(d.$**s-***Parent-sort-identifier*, **sort**$).derivedDataType_1$

The function $isCompatibleTo_1$ determines if a *Sort-reference-identifier* is compatible to the other.

$isCompatibleTo_1(id_1: Sort\text{-}reference\text{-}identifier, id_2: Sort\text{-}reference\text{-}identifier): BOOLEAN =_{def}$
let $d_1 = getEntityDefinition_1(id_1,$ **sort**$)$ **in**
let $d_2 = getEntityDefinition_1(id_2,$ **sort**$)$ **in**
$d_1 = d_2 \vee isSuperType_1(d_2, d_1)$
endlet
endlet

2.1.6.4 Function Definitions on AS0

The function $name_0$ gets the name of a given entity or reference.

$name_0(d: ENTITYDEFINITION_0 \cup REFERENCE_0$): <name> $=_{def}$
case *d* **of**
| $REFERENCE_0$ => $d.referenceName_0$
| $ENTITYDEFINITION_0$ => $d.entityName_0$
otherwise *undefined*
endcase

The function $entityName_0$gets the name of a given entity definition.

$entityName_0(ed: ENTITYDEFINITION_0)$: <name> $=_{def}$
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>
| <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 definition> => *ed*.**s-**<name>
| <composite state> => *ed*.**s-**<composite state heading>.**s-**<name>
| <data type definition> => *ed*.**s-**<data type heading>.**s-**<name>
| <signal definition item> => *ed*.**s-**<name>
| <exception definition item> => *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>
| *FORMALCONTEXTPARAMETER*$_0$=> *ed*.**s-**<name>
| <syntype definition> =>
let *s*=*ed*.**s-implicit in**
if *s*∈ <syntype definition gen syntype> **then** *s*.**s-**<syntype<name>
else *s*.**s-**<data type heading>.**s-**<data type<name>
endif
endlet endcase

The function *referenceName*$_0$ gets the name of a given reference.

referenceName$_0$(*ref*: *REFERENCE*$_0$): <name> =$_{def}$
case *ref* **of**
| *TYPEREFERENCE*$_0$=>*ref*. **s-**<identifier>.**s-**<name>
| <textual operation reference> => *ref*.**s-**<operation heading>.**s-**<operation name>
otherwise *ref*.**s-**<name>
endcase

The function *kind*$_0$gets the name of a given entity definition or reference.

kind$_0$(*d*: *ENTITYDEFINITION*$_0$∪ *REFERENCE*$_0$): *ENTITYKIND*$_0$ =$_{def}$
case *ed* **of**
| *REFERENCE*$_0$ => *d.referenceKind*$_0$
| *ENTITYDEFINITION*$_0$=> *d*. *entityKind*$_0$
endcase

The function *entityKind*$_0$gets the name of a given entity definition.

entityKind$_0$(*ed*: *ENTITYDEFINITION*$_0$): *ENTITYKIND*$_0$ =$_{def}$
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 definition> => **state**
| <composite state type definition> ∪ <composite state type context parameter> => **state type**
| <procedure definition> ∪ <procedure context parameter> => **procedure**
| <signal definition item> ∪ <signal context parameter gen 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 gen name> => **interface**
| <textual gate definition> ∪ <gate context parameter> => **gate**
| <exception definition item> ∪ <exception context parameter gen name> => **exception**
| <timer definition item> ∪ <timer context parameter gen 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 gen name> => **synonym**

| <variable context parameter gen name> ∪ <variables of sort>=>**variable**
| <remote variable context parameter gen name> ∪ <remote variable definition gen name> =>
remote variable
otherwise *undefined*
endcase

The function *referenceKind*$_0$ gets the entity kind of a specified reference.

referenceKind$_0$(*ref*: *REFERENCE*$_0$): *ENTITYKIND*$_0$=$_{def}$
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**
| <signal reference> => **signal**
| <data type reference> => **type**
| <procedure reference> => **procedure**
| <interface reference> => **interface**
| <textual operation reference> => *ref*.**s-**<operation kind>
otherwise *undefined*
endcase

The function *qualifier*$_0$ gets the qualifier specified in an entity definition or a reference.

qualifier$_0$(*d*: *ENTITYDEFINITION*$_0$∪*REFERENCE*$_0$): <qualifier> =$_{def}$
take({*q*∈ <qualifier>: *q.parentAS0*. *parentAS0* = *d*})

The function *surroundingScopeUnit*$_0$ gets the surrounding scope unit for an node in AS0.

surroundingScopeUnit$_0$(*d*: *DefinitionAS0*): *SCOPEUNIT*$_0$ =$_{def}$
if *d* ∈ <referenced definition> **then**
parentAS0ofKind(*d.referencedBy*$_0$, *SCOPEUNIT*$_0$)
else
parentAS0ofKind(*d*, *SCOPEUNIT*$_0$)
endif

2.1.6.5 Lexis

The following lexical items are still 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> :: ()

2.2 Visibility Rules, Names and Identifiers

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*
Exception-handler-name	=	*Name*
Connector-name	=	*Name*
State-entry-point-name	=	*Name*
State-exit-point-name	=	*Name*
Channel-name	=	*Name*
Variable-name	=	*Name*

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>

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*) =>
case *x.parentAS0* **of**
| <transition option> ∪ <task> ∪ <answer part> ∪ <decision> => **mk-***Informal-text*(*x*)
otherwise mk-*Name*(*x*)
endcase

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*
	|	*Expanded-sort*
	|	*Reference-sort*
Sort-identifier	=	*Identifier*
Syntype-identifier	=	*Identifier*
Expanded-sort	::	*Sort-identifier*
Reference-sort	::	*Sort-identifier*
Timer-identifier	=	*Identifier*
Gate-identifier	=	*Identifier*
Exception-identifier	=	*Identifier*
Composite-state-type-identifier	=	*Identifier*
Channel-identifier	=	*Identifier*
Literal-identifier	=	*Identifier*
Operation-identifier	=	*Identifier*
Variable-identifier	=	*Identifier*

Conditions on Abstract Syntax

$\forall d, d' \in \mathit{ENTITYDEFINITION}_1$: $d.entityKind_1 = d'.entityKind_1 \wedge d \neq d' \Rightarrow d.identifier_1 \neq d'.identifier_1$

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

Concrete Syntax

<identifier> :: <qualifier> <name>
<qualifier> = <path item>*

Conditions on Concrete Syntax

$\forall d1, d2 \in \mathit{ENTITYDEFINITION}_0$:
($d1 \neq d2 \wedge d1.entityKind_0 = d2.entityKind_0 \wedge d1.surroundingScopeUnit_0 = d2.surroundingScopeUnit_0) \Rightarrow$
(($d1 \notin$ <operation signature>$\Rightarrow d1.entityName_0 \neq d2.entityName_0) \wedge$
($d1 \in$ <operation signature>$\Rightarrow \neg isSameOperationSignature_0(d1, d2) \vee d1.entityName_0 \neq d2.entityName_0$))

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* = $fullIdentifier_0(i)$ **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.

$fullIdentifier_0(i$:<identifier>): <identifier> $=_{\text{def}}\ i.refersto_0.identifier_0$

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

$refersto_0(i$:<identifier>): $\mathit{ENTITYDEFINITION}_0 =_{\text{def}}\ getEntityDefinition_0(i, idKind_0(i))$

For any given entity definition in AS0, the function *identifier*$_0$ returns its identifier with full qualifier.

identifier$_0$(*def*: *ENTITYDEFINITION*$_0$): <identifier> =$_{def}$
<identifier>(*def.fullQualifier*$_0$, *def.entityName*$_0$)

The function *fullQualifier*$_0$ is used to get the full qualifier for an entity definition.

fullQualifier$_0$(*d*: *ENTITYDEFINITION*$_0$): <qualifier> =$_{def}$
let *su* = *d*. *surroundingScopeUnit*$_0$ **in**
if *su* = *undefined* **then** *empty*
else *su.fullQualifier*$_0$^<path item>(*su.entityKind*$_0$, *su.entityName*$_0$)
endlet

The function *getEntityDefinition*$_0$ is used to get the definition that the given identifier refers to.

getEntityDefinition$_0$(*id*:<identifier>, *ek*: *ENTITYKIND*$_0$): *ENTITYDEFINITION*$_0$ =$_{def}$
if *ek*∈ {**operator**, **literal**, **method**} **then**
resolutionByContext$_0$(*id*)
else
let *su* = *getStartingScopeUnit*$_0$(*id*, *id.surroundingScopeUnit*$_0$) **in**
if *su* = *undefined* **then** *undefined*
else *resolutionByContainer*$_0$(*su*, *id*, *ek*)
endif
endlet
endif

The function *resolutionByContainer*$_0$ binds an <identifier> to a definition through resolution by container.

resolutionByContainer$_0$(*su*: *SCOPEUNIT*$_0$, *id*:<identifier>, *ek*:*ENTITYKIND*$_0$): *ENTITYDEFINITION*$_0$ =$_{def}$
let *d1=bindInLocalDefinition*$_0$(*su*, *id*, *ek*) **in**
if *d1*≠ *undefined* **then** *d1*
else let *d2=bindInBaseType*$_0$(*su*, *id*, *ek*) **in**
if *d2*≠ *undefined* **then** *d2*
else let *d3=bindInUsedPackage*$_0$(*su*.**s-**<package use clause>, *id*, *ek*) **in**
if *d3*≠ *undefined* **then** *d3*
else let *d4=bindInLocalInterface*$_0$(*su.localInterfaceDefinitionSet*$_0$, *id*, *ek*) **in**
if *d4*≠ *undefined* **then** *d4*
else let *su'=su.surroundingScopeUnit*$_0$ **in**
if *su'* ≠ *undefined* **then** *resolutionByContainer*$_0$(*su'*, *id*, *ek*)
else *undefined*
endif endlet
endif endlet
endif endlet
endif endlet

The function *bindInLocalDefinition*$_0$ is used to search in the given scope unit to determine if there exists a local entity definition for the specified identifier.

bindInLocalDefinition$_0$(*su*: *SCOPEUNIT*$_0$, *id*: <identifier>, *ek*: *ENTITYKIND*$_0$): *ENTITYDEFINITION*$_0$=$_{def}$
let *d* = *take*({*d*∈ *ENTITYDEFINITION*$_0$:
d.surroundingScopeUnit$_0$ = *su* ∧ *isSameEntityName*$_0$(*id*.**s-**<name>, *d*)∧
isConsistentKindTo$_0$(*d.entityKind*$_0$, *ek*)∧*isVisibleIn*$_0$(*d*, *id.surroundingScopeUnit*$_0$)}) **in**
if *d* ≠ *undefined* **then** *d*
else let *rd* = *take*({*rd* ∈ *REFERENCE*$_0$: *rd.surroundingScopeUnit*$_0$ = *su* ∧
rd. *referencedDefinition*$_0$.*entityName*$_0$= *id*.**s-**<name>∧
isConsistentKindTo$_0$(*rd.referencedDefinition*$_0$.*entityKind*$_0$, *ek*) ∧
isVisibleIn$_0$(*rd*, *id.surroundingScopeUnit*$_0$))}) **in**

```
            if rd ≠ undefined then rd.referencedDefinition0
            else undefined
            endif endlet
        endif
    endlet
```

The function *bindInBaseType*$_0$ 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 *bindInUsedPackage*$_0$ 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 *bindInLocalInterface*$_0$ 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 *isSameEntityName*$_0$ 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 )∨
    (d.entityKind0 = literal ∧ d∈ <name class literal> ∧
        isInRegularExpr0(n, d.s-<regular expression>.s-<or partial regular expression>-seq))
```

For a given identifier (left most path item may be omitted), the function *getStartingScopeUnit*$_0$ 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 *getStartingSuInUsedPackage*$_0$ finds the starting scope unit in the packages list.

getStartingSuInUsedPackage$_0$(*id*: <identifier>, *pdl*:<package definition>*): *ScopeUnit*$_0$=$_{\text{def}}$
if *pdl =empty* **then** *undefined*
elseif *isQualifierMatched*$_0$(*id*.**s-**<qualifier>, *pdl.head*) **then** *pdl.head*
else *getStartingSuInUsedPackage*$_0$(*id*, *pdl.tail*)
endif

The function *getStartingSuInInterface*$_0$ finds the starting scope unit in the interface list.

getStartingSuInInterface$_0$(*id*: <identifier>, *ifds*:<interface definition>-**set**): *ScopeUnit*$_0$=$_{\text{def}}$
if *ifds* = ∅ **then** *undefined*
else let *d* = *ifds.take* **in**
if *isQualifierMatched*$_0$(*id*.**s-**<qualifier>, *d*) **then** *d*
else *getStartingSuInInterface*$_0$(*id*, *ifds* \ {*d*})
endif endlet
endif

The function *isDefinedIn*$_0$ determines if an entity definition is defined in a given scope unit.

isDefinedIn$_0$(*ed*: *EntityDefinition*$_0$, *su*: *ScopeUnit*$_0$): *Boolean*=$_{\text{def}}$
if (*su* = *undefined* ∨ *ed* =*undefined*)**then** *False*
else
let *su′* = *ed.surroundingScopeUnit*$_0$ **in**
su = *su′* ∨
isVisibleInInterface$_0$(*ed*, *su*) ∨
isVisibleInDataType$_0$(*ed*, *su*) ∨
isVisibleThroughBaseType$_0$(*ed*, *su*)
endlet
endif

The function *isVisibleIn*$_0$ determines if an entity definition is visible in a given scope unit.

isVisibleIn$_0$(*ed*: *EntityDefinition*$_0$, *su*: *ScopeUnit*$_0$): *Boolean*=$_{\text{def}}$
isDefinedIn$_0$(*ed*, *su*) ∨
isVisibleThroughUsedPackage$_0$(*ed*, *su*) ∨
isVisibleIn$_0$(*ed*, *su.surroundingScopeUnit*$_0$)

The function *isVisibleInInterface*$_0$ determines if the scope unit contains an <interface definition> which is the defining context of the entity.

isVisibleInInterface$_0$(*ed*: *EntityDefinition*$_0$, *su*: *ScopeUnit*$_0$): *Boolean*=$_{\text{def}}$
(*ed.surroundingScopeUnit*$_0$∈ *su.localInterfaceDefinitionSet*$_0$)

The function *isVisibleInDataType*$_0$ determines if the scope unit contains a <data type definition> which is the defining context of the entity.

isVisibleInDataType$_0$(*ed*: *EntityDefinition*$_0$, *su*: *ScopeUnit*$_0$): *Boolean*=$_{\text{def}}$
(*ed.surroundingScopeUnit*$_0$∈ *su.localDataTypeDefinitionSet*$_0$) ∧
(*ed*∈ <literal signature> ∪ <operation signature> ⇒ *ed.isPublic*$_0$)

The function *isVisibleThroughBaseType*$_0$ determines if the entity is visible through the base type of the scope unit.

isVisibleThroughBaseType$_0$(*ed*: *EntityDefinition*$_0$, *su*: *ScopeUnit*$_0$): *Boolean*=$_{\text{def}}$
let *spec* = *su*. *specialization*$_0$ **in**
if *spec* = *undefined* **then** *False*
else (∃*btd*∈ *TypeDefinition*$_0$: *isDirectSubType*$_0$(*su*, *btd*) ∧ *isVisibleIn*$_0$(*ed*, *btd*) ∧

(*ed* ∈ <literal signature>∪<operation signature> ⇒
¬*isPrivate*$_0$(*ed*)∧¬ *isRenamedBy*$_0$(*ed*, *spec*))∧
ed∈ *FORMALCONTEXTPARAMETER*$_0$⇒¬*isBoundToActualContextPara*$_0$(*ed*, *spec*))
endif
endlet

The function *isVisibleThroughUsedPackage*$_0$ determines if an entity definition is visible through used packages.

isVisibleThroughUsedPackage$_0$(*ed*: *ENTITYDEFINITION*$_0$, *su*: *SCOPEUNIT*$_0$): *BOOLEAN* =$_{def}$
let *ucs* = *su*.**s**-<package use clause>.*toSet* **in**
if *ucs* = ∅ **then** *False*
else (∃*uc*∈*ucs*: *ed.surroundingScopeUnit*$_0$= *uc.usedPackage*$_0$∧*isVisibleThroughUseClause*$_0$(*ed*, *uc*))
endif
endlet

The function *isBoundToActualContextPara*$_0$ determines if a <formal context parameter> is bound to an <actual context parameter> in a <specification>.

isBoundToActualContextPara$_0$(*fcp*: *FORMALCONTEXTPARAMETER*$_0$, *spec*: <specialization>): *BOOLEAN* =$_{def}$
(∃*acp*∈ <actual context parameter>:
acp. *parentAS0.parentAS0* = *spec*∧*isContextParameterCorresponded*$_0$(*fcp*, *acp*))

The function *isVisibleThroughUseClause*$_0$ determines if an entity definition is visible through the <package interface> and <package use clause>.

isVisibleThroughUseClause$_0$(*ed*: *ENTITYDEFINITION*$_0$, *uc*:<package use clause>): *BOOLEAN* =$_{def}$
let *pd*=*uc.usedPackage*$_0$ **in**
let *pi* = *pd*. **s**-<package heading>.**s**-<package interface> **in**
(*pi* = *undefined*∨
∃*ds*∈ <definition selection>: *ds.parentAS0*= *pi*∧
isMentionedInDefSelection$_0$(*ed*, *ds*, *pd*)) ∧
(*uc*.**s**-<definition selection>-**seq** = *empty* ∨
∃*ds*∈<definition selection>: *ds.parentAS0* = *uc* ∧
isMentionedInDefSelection$_0$(*ed*, *ds*, *pd*))
endlet

The function *isMentionedInDefSelection*$_0$ determines if an entity is mentioned in a <definition selection>.

isMentionedInDefSelection$_0$(*ed*: *ENTITYDEFINITION*$_0$, *ds*:<definition selection>, *pd*:<package definition>):
BOOLEAN =$_{def}$
(*ds*.**s**-<name> = *ed*. *entityName*$_0$ ∧
(*ds*.**s**-<selected entity kind> ≠*undefined*⇒ *ds*.**s**-<selected entity kind> = *ed*. *entityKind*$_0$))∨
(*ed.entityKind*$_0$= **signal** ∧ *ds*.**s**-<selected entity kind> = **signallist** ∧
(∃*sld*∈<signal list definition>: *sld.surroundingScopeUnit*$_0$= *pd*∧
sld. *entityName*$_0$= *ds*.**s**-<name> ∧
(∃*sigId*∈ *sld.signalSet*$_0$: *getEntityDefinition*$_0$(*sigId*, **signal**) = *ed*)))

The function *isConsistentKindTo*$_0$ is used to determine if the first entity kind is consistent to the second one.

isConsistentKindTo$_0$(*t1*: *ENTITYKIND*$_0$, *t2*: *ENTITYKIND*$_0$): *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 *isQualifierMatched*$_0$ is used to determine if the given <qualifier> is the same as the rightmost part of the full <qualifier> denoting the given scope unit

isQualifierMatched$_0$(*q*: <qualifier>, *su*: *SCOPEUNIT*$_0$): *BOOLEAN* =$_{def}$
if *q* = *undefined* **then** *True*
elseif *su*∈ <compound statement> **then** *False*

else let *q′* = *su.fullQualifier*$_0$ **in**
let *seq1* = *q*.**s-**<path item>**-seq in**
let *seq2* = *q′* ⌢<path item>(*su.entityKind*$_0$, *su.entityName*$_0$) **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 *resolutionByContext*$_0$ is used to bind all <name>s of entity kind **operator**, **method** and **literal** to their corresponding entity definitions.

resolutionByContext$_0$(*id*:<identifier>):*ENTITYDEFINITION*$_0$=$_{\text{def}}$
let *bl* = *take*(*getBindingListSet*$_0$(*id.contextOfIdentifier*$_0$)) **in**
getDefinitionInBindingList$_0$(*id*.**s-**<name>, *bl*)
endlet

The function *contextOfIdentifier*$_0$ gets the context for resolving the identifier.

contextOfIdentifier$_0$(*id*:<identifier>):*CONTEXT*$_0$=$_{\text{def}}$
if (∃*exp*∈ <expression>: *isAncestorAS0*(*exp*, *id*)) **then**
contextOfExp$_0$(*parentAS0ofKind*(*id*, <expression>))
else *undefined*

The function *getBindingListSet*$_0$ is used to bind all <name>s of entity kind **operator**, **method** and **literal** in the context to their corresponding entity definitions.

getBindingListSet$_0$(*c*: *CONTEXT*$_0$): *BINDINGLIST*$_0$**-set** =$_{\text{def}}$
let *nameList* = *c.nameList*$_0$ **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** ∅
elseif (∀*bl1*, *bl2*∈ *resultSet*: ∀*i*∈ 1..*bl1.length*: *bl1*[*i*].**s-***ENTITYDEFINITION0.entityKind0*≠ **literal** ⇒
representSameDynamicOpSig0(*bl1*[*i*].**s-***ENTITYDEFINITION0*, *bl2*[*i*].**s-***ENTITYDEFINITION0*))
then *resultSet*
else ∅
endif
endlet

representSameDynamicOpSig$_0$(*os1*:<operation signature>, *os2*: <operation signature>):*BOOLEAN*=$_{\text{def}}$
(*os1.superCounterpart*$_0$ ∪ {*os1*}) ∩ (*os2.superCounterpart*$_0$ ∪ {*os2*}) ≠ ∅

The function *nameList*$_0$ gets all the <name>s of entity kind **operator**, **method** and **literal** appeared in the context.

nameList$_0$(*c*: *CONTEXT*$_0$): <name>*=$_{\text{def}}$
case *c* **of**
|<assignment>=>*c*.**s-**<variable>.*nameListInVariable*$_0$⌢*c*.**s-**<expression>.*nameListInExpression0*
|<decision>=>*c.nameListInDecision0*
|<expression>=>*c.nameListInExpression0*
otherwise *empty*
endcase

The function *nameListInExpression*$_0$ gets all the <name>s of entity kind **operator**, **method** and **literal** appeared in the <expression>.

nameListInExpression$_0$(*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>(*e1*, *op*, *e2*)=>
e1.nameListInExpression0^*op*^*e2.nameListInExpression0*
|<equality expression>=>
exp. **s-**<expression>*.nameListInExpression0*^ *exp*.**s2-**<expression>*.nameListInExpression0*
|<expression gen primary>=>*exp*.**s-implicit**^*exp*.**s-**<primary>.*nameListInPrimary*$_0$
endcase

The function *nameListInPrimary*$_0$ gets all the <name>s of entity kind **operator**, **method** and **literal** appeared in the <primary>.

nameListInPrimary$_0$(*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 *nameListInOperationApplication*$_0$ gets all the <name>s of entity kind **operator**, **method** and **literal** appeared in the <operation application>.

nameListInOperationApplication$_0$(*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 *nameListInExtendedPrimary*$_0$ gets all the <name>s of entity kind **operator**, **method** and **literal** appeared in the <extended primary>.

nameListInExtendedPrimary$_0$(*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>*.nameListInActualParameterList0*
endcase

The function *nameListInActualParameterList*$_0$ gets all the <name>s of entity kind **operator**, **method** and **literal** appeared in the actual parameter list.

nameListInActualParameterList$_0$(*el*: <expression>*): <name>*=$_{def}$
if *el* = *empty* **then** *empty*
else
el.head.nameListInExpression$_0$^*el.tail.nameListInActualParameterList*$_0$
endif

The function *nameListInVariable*$_0$ gets all the <name>s of entity kind **operator**, **method** and **literal** appeared in <variable>.

nameListInVariable$_0$(*v*:<variable>):<name>*=$_{def}$
if *v*∈ <indexed variable> **then**
v.**s-**<variable>.*nameListInVariable*$_0$^*v*.**s-**<actual parameter>**-seq**.*nameListInActualParameterList*$_0$
elseif *v*∈ <field variable> **then**
v.**s-**<variable>.*nameListInVariable*$_0$
else *empty*
endif

The function *nameListInDecision*$_0$ gets all the <name>s of entity kind **operator**, **method** and **literal** appeared in the <decision>.

nameListInDecision$_0$(*d*: <decision>):<name>*=$_{def}$
d.**s-**<question>.*nameListInExpression*$_0$^ *d.rangeConditionList*$_0$.*nameListInRangeConditions*$_0$

The function *nameListInRangeConditions*$_0$ gets all the <name>s of entity kind **operator**, **method** and **literal** appeared in the <range condition> list.

nameListInRangeConditions$_0$(*rcl*:<range condition>*):<name>*=$_{def}$
if *rcl* = *empty* **then** *empty*
else *rcl.head.nameListInRangeList*$_0$^*rcl.tail.nameListInRangeConditions*$_0$

The function *nameListInRangeList*$_0$ gets all the <name>s of entity kind **operator**, **method** and **literal** appeared in the <range> list.

nameListInRangeList$_0$(*rl*:<range>*):<name>*=$_{def}$
if *rl* = *empty* **then** *empty*
else *rl.head.nameListInRange*$_0$^*rl.tail.nameListInRangeList*$_0$

The function *nameListInRange*$_0$ gets all the <name>s of entity kind **operator**, **method** and **literal** appeared in the <range>.

nameListInRange$_0$(*r*: <range>):<name>*=$_{def}$
case *r* **of**
|<closed range>(*c1*, *c2*)=>
c1.nameListInExpression$_0$^*c2.nameListInExpression*$_0$
|<open range>(*c1*, *n*, *c2*)=>
c1.nameListInExpression$_0$^<*n*>^*c2.nameListInExpression*$_0$
endcase

Each element in the *possibleBindingListSet*$_0$ represents a possible resolution for the given name list.

possibleBindingListSet$_0$(*n*: <name>*):*BINDINGLIST*$_0$**-set** =$_{def}$
{*b*∈ *BINDINGLIST*$_0$: *b.length* = *n.length*∧
∀*i*∈ 1.. *b.length*:*b*[*i*].**s-**<name>= *n*[*i*]∧ *b*[*i*].**s-***ENTITYDEFINITION*$_0$∈ *n*[*i*].*possibleDefinitionSet*$_0$}

The function *isSatisfyStaticCondition*$_0$ determines if the binding violates any static sort constraints in the context.

isSatisfyStaticCondition$_0$(*bl*:*BINDINGLIST*$_0$, *c*:*CONTEXT*$_0$): *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>.

isSatisfyAssignmentCondition$_0$(*bl*:*BINDINGLIST*$_0$, *ass*: <assignment>): *BOOLEAN*=$_{def}$

let *varSort*= *getVariableSort*$_0$(*ass*.**s-**<variable>) **in**
let *expSort*= *getStaticSort*$_0$(*ass*.**s-**<expression>, *bl*) **in**
(*isSortCompatible*$_0$(*varSort*, *expSort*) ∨ *isSortCompatible*$_0$(*expSort*, *varSort*))∧
(*ass*.**s-**<variable>∈ <indexed variable>⇒*isSatisfyIndexVariableCondition*$_0$(*bl*, *ass*.**s-**<variable>)
endlet

The function *isSatisfyIndexVariableCondition*$_0$ determines if the binding violates any static sort constraints in the <indexed variable>.

isSatisfyIndexVariableCondition$_0$(*bl*, *var*:<indexed variable>): *BOOLEAN*=$_{\text{def}}$
let *acp*=*var*.**s-**<actual parameter>**-seq in**
isSortCompatible$_0$(*getStaticSort*$_0$(*acp*[1],*bl*), *getIndexSort*$_0$(*getVariableSort*$_0$(*var*)))∧
(*var*.**s-**<variable>∈ <indexed variable>⇒*isSatisfyIndexVariableCondition*$_0$(*bl*, *var*.**s-**<variable>))
endlet

Get the static sort of a <variable>.

getVariableSort$_0$(*var*:<variable>):<sort>=$_{\text{def}}$
case *var* **of**
| <identifier>=>*getEntityDefinition*$_0$(*var*, **variable**).*parentAS1*.**s-**<sort>
| <indexed variable>=>*getItemSort*$_0$(*getVariableSort*$_0$(*var*.**s-**<variable>))
| <field variable>=>*getFieldSort*$_0$(*getVariableSort*$_0$(*var*.**s-**<variable>), *var*.**s-**<field name>)
endcase

The function *getItemSort*$_0$ gets the item sort of a <sort> that is a subtype of the predefined sort String or Array.

getItemSort$_0$(*s*: <sort>):<sort>=$_{\text{def}}$
let *d*=*getEntityDefinition*$_0$(*s*, **sort**).*derivedDateType*$_0$ **in**
if *d*.*specialization*$_0$ = *undefined* **then** *undefined*
elseif *d*.*specialization*$_0$.**s-**<base type>.**s-**<name> = "String" **then** *d*.*actualContextParameterList*$_0$[1]
elseif *d*.*specialization*$_0$.**s-**<base type>.**s-**<name> = "Array" **then** *d*.*actualContextParameterList*$_0$[2]
else *undefined*
endif
endlet

The function *getIndexSort*$_0$ gets the index sort of a <sort> that is a subtype of the predefined sort String or Array.

getIndexSort$_0$(*s*: <sort>):<sort>=$_{\text{def}}$
let *d*=*getEntityDefinition*$_0$(*s*, **sort**).*derivedDateType*$_0$ **in**
if *d*.*specialization*$_0$ = *undefined* **then** *undefined*
elseif *d*.*specialization*$_0$.**s-**<base type>.**s-**<name> = "String" **then**
<identifier>(<qualifier>(<name>("Predefined")),"Interger")
elseif *d*.*specialization*$_0$.**s-**<base type>.**s-**<name> = "Array" **then** *d*.*actualContextParameterList*$_0$[1]
else *undefined*
endif
endlet

The function *getFieldSort*$_0$ gets the field sort of a field name in the <data type definition> referred by the given <sort>.

getFieldSort$_0$(*s*: <sort>, *n*:<name>):<sort>=$_{\text{def}}$
let *d*=*getEntityDefinition*$_0$(*s*, **sort**).*derivedDateType*$_0$ **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 *isSatisfyStaticCondition*$_0$ determines if the binding violates any static sort constraints in the <decision>.

isSatisfyDecisionCondition$_0$(*bl*:*BINDINGLIST*$_0$, *d*: <decision>): *BOOLEAN*=$_{\text{def}}$
let *q*=*d*.**s-**<question>, *rcs*=*d*.*rangeConditionList*$_0$.*toSet* **in**

isSatisfyExpressionCondition$_0$(*bl*, *q*)∧
(∀*rc1*∈*rcs*:∀*ce1*∈<constant expression>:*isAncestorAS1*(*rc1*, *ce1*)⇒
isSatisfyExpressionCondition$_0$(*bl*, *ce1*)∧
isSortCompatible$_0$(*getStaticSort*$_0$(*ce1*, *bl*), *getStaticSort*$_0$(*q*, *bl*)∧
(∀*rc2*∈*rcs*: ∀*ce2*∈<constant expression>:*isAncestorAS1*(*rc2*, *ce2*)⇒
(*isSortCompatible*$_0$(*getStaticSort*$_0$(*ce1*, *bl*), *getStaticSort*$_0$(*ce2*, *bl*))∨
isSortCompatible$_0$(*getStaticSort*$_0$(*ce2*, *bl*), *getStaticSort*$_0$(*ce1*, *bl*))))
endlet

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

isSatisfyExpressionCondition$_0$(*bl*:*BINDINGLIST*$_0$, *exp*: <expression>): *BOOLEAN*=$_{\text{def}}$
case *exp* **of**
|<create expression>=>*isSatisfyCreateCondition*$_0$(*bl*, *exp*)
|<value returning procedure call>=>
if *exp*∈ <procedure call body> **then** *isSatisfyProcedureCallBodyCondition*$_0$(*bl*, *exp*)
else *isSatisfyRemoteProcCallBodyCondition*$_0$(*bl*, *exp*)
endif
|<range check expression> => *isSatisfyRangeCheckCondtion*$_0$(*bl*, *exp*)
|<equality expression>=> *isSatisfyEqualityExpCondtion*$_0$(*bl*, *exp*)
|<binary expression>=>
let *opDef* = *getDefinitionInBindingList*$_0$(*exp*.**s-implicit**, *bl*) **in**
let *fpl* = *opDef.operationParameterSortList*$_0$ **in**
fpl.length = 2∧
isSortCompatible$_0$(*getStaticSort*$_0$(*exp*.**s-**<expression>, *bl*), *fpl*[1])∧
isSortCompatible$_0$(*getStaticSort*$_0$(*exp*.**s2-**<expression>, *bl*), *fpl*[2])∧
isSatisfyExpressionCondition0 (*bl*, *exp*.**s-**<expression>)∧
isSatisfyExpressionCondition0 (*bl*, *exp*.**s2-**<expression>)
endlet
|<expression gen primary>=>
if *exp*.**s-implicit** = *undefined* **then**
isSatisfyPrimaryCondition$_0$(*bl*, *pr*)
else
let *opDef* = *getDefinitionInBindingList*$_0$(*exp*.**s-implicit**, *bl*) **in**
let *fpl* = *opDef.operationParameterSortList*$_0$ **in**
fpl.length = 1∧*isSortCompatible*$_0$(*getStaticSort*$_0$(*pr*, *bl*), *fpl*[1])∧
isSatisfyPrimaryCondition$_0$(*bl*, *pr*)
endlet
endif
endcase

The function *isSatisfyCreateCondition*$_0$ determines if the binding violates any static sort constraints in the <expression>.

isSatisfyCreateCondition$_0$(*bl*: *BINDINGLIST*$_0$, *ce*: <create expression>):*BOOLEAN* =$_{\text{def}}$
let *def* = *ce.getCreatedAgentDefinition*$_0$ **in**
if *def* =*undefined* **then** *False*
else
let *fpl* = *def.agentFormalParameterList*$_0$ **in**
let *apl* = *ce.actualParameterList*$_0$ **in**
fpl.length=*apl.length* ∧
(∀*i*∈ 1..*fpl.length*: *isSortCompatible*$_0$(*getStaticSort*$_0$(*apl*[*i*], *bl*), *fpl*[*i*].*parentAS1*.**s-**<sort>)∧
isSatisfyExpressionCondition$_0$(*bl*, *apl*[*i*]))
endlet
endif endlet

The function *getCreateExpSort*$_0$ gets the static sort of <create expression>.

getCreateExpSort$_0$(*ce*: <create expression>): <sort>=$_{def}$
let *def* = *ce*.*getCreatedAgentDefinition*$_0$ **in**
if *def* =*undefined* **then** Pid
else *def*.*identifier*$_0$
endif endlet

The function *isSatisfyProcedureCallBodyCondition*$_0$ determines if the binding violates any static sort constraints in the <prcodure call body>.

isSatisfyProcedureCallBodyCondition$_0$(*bl*: *BINDINGLIST*$_0$, *body*: <procedure call body>):*BOOLEAN*=$_{def}$
let *apl* = *body*.*actualParameterList*$_0$ **in**
let *fpsl*=*calledProcedure*$_0$.*formalParameterSortList*$_0$ **in**
fpsl.*length*=*apl*.*length* ∧
(∀*i*∈ 1..*fpsl*.*length*: *isSortCompatible*$_0$(*getStaticSort*$_0$(*apl*[*i*], *bl*), *fpsl*[*i*)∧
isSatisfyExpressionCondition$_0$(*bl*, *apl*[*i*]))
endlet

The function *getProcCallBodySort*$_0$ gets the static sort of <procedure call body>.

getProcCallBodySort$_0$(*body*: <procedure call body>): <sort>=$_{def}$
case *body*.**s-implicit of**
| <identifier>=>*getEntityDefinition*$_0$(*body*.**s-implicit**, **procedure**).*procedureResult*$_0$
pd.**s-**<procedure heading>.**s-**<procedure result>
| <type expression>=> *body*.**s-implicit**.*baseType*$_0$.*procedureResult*$_0$
endcase

The function *procedureResult*$_0$ gets the result sort of a <prcedure definition>.

procedureResult$_0$(*pd*: <procedure definition>):<sort>=$_{def}$
if *pd*.**s-**<procedure heading>.**s-**<procedure result>≠*undefined* **then**
pd.**s-**<procedure heading>.**s-**<procedure result>
elseif *pd*.*specialization*$_0$ ≠ *undefined* **then**
pd.*baseType*$_0$.*procedureResult*$_0$
else *undefined*
endif

The function *getRemoteProcCallBodySort*$_0$ gets the static sort of <remote procedure call body>.

getRemoteProcCallBodySort$_0$(*body*: <remote procedure call body>, *bl*: *BINDINGLIST*$_0$): <sort>=$_{def}$
getEntityDefinition$_0$(*body*.**s-**<remote procedure<identifier>,**remote procedure**).*procedureResult*$_0$

The function *isSatisfyRemoteProcCallBodyCondition*$_0$ determines if the binding violates any static sort constraints in the <remote prcodure call body>.

isSatisfyRemoteProcCallBodyCondition$_0$(*bl*: *BINDINGLIST*$_0$, *body*: <remote procedure call body>):*BOOLEAN* =$_{def}$
let *rpd* = *getEntityDefinition*$_0$(*body*.**s-**<remote procedure <identifier>,**remote procedure**) **in**
let *fpsl*= *rpd*.*formalParameterSortList*$_0$ **in**
let *apl* = *body*.*actualParameterList*$_0$ **in**
fpsl.*length*=*apl*.*length* ∧
(∀*i*∈ 1..*fpsl*.*length*: *isSortCompatible*$_0$(*getStaticSort*$_0$(*apl*[*i*], *bl*), *fpsl*[*i*])∧
isSatisfyExpressionCondition$_0$(*bl*, *apl*[*i*]))
endlet

The function *operationResult*$_0$ gets the result sort of an <operation signature>.

operationResult$_0$(*os*:<operation signatures>):<sort>=$_{def}$
if *os*∈ *PREDEFINEDOPERATION*$_0$ **then** *os*. *getPredefinedOpResult*$_0$
else *os*.**s-**<result>
endif

The function *isSatisfyRangeCheckCondtion*$_0$ determines if the binding violates any static sort constraints in the <range check expression>.

isSatisfyRangeCheckCondtion$_0$(*bl*: *BINDINGLIST*$_0$, *rce*: <range check expression>): *BOOLEAN*$=_{def}$
if *rce*.**s-implicite**$\in$ <sort> **then**
isSatisfyExpressionCondition$_0$(*bl*, *rce*.**s-**<expression>)$\wedge$
isSameSort$_0$(*getStaticSort*$_0$(*rce*.**s-**<expression>, *bl*), *rce*.**s-implicite**$\in$ <sort>)
else
isSatisfyExpressionCondition$_0$(*bl*, *rce*.**s-**<expression>)$\wedge$
isSameSort$_0$(*getStaticSort*$_0$(*op2*, *bl*), *rce*.**s-implicite**.**s-**<sort<identifier>)
endif

The function *isSatisfyEqualityExpCondtion*$_0$ determines if the binding violates any static sort constraints in the <equality expression>.

isSatisfyEqualityExpCondtion$_0$(*bl*: *BINDINGLIST*$_0$, *eq*: <equality expression>): *BOOLEAN*$=_{def}$
isSatisfyExpressionCondition$_0$(*bl*, *eq*.**s-**<expression>)$\wedge$
isSatisfyExpressionCondition$_0$(*bl*, *eq*.**s2-**<expression>)$\wedge$
(*isSortCompatible*$_0$(*getStaticSort*$_0$(*eq*.**s-**<expression>, *bl*), *getStaticSort*$_0$(*eq*.**s2-**<expression>, *bl*))$\vee$
isSortCompatible$_0$(*getStaticSort*$_0$(*eq*.**s2-**<expression>, *bl*), *getStaticSort*$_0$(*eq*.**s-**<expression>, *bl*)))

The function *isSatisfyPrimaryCondition*$_0$ determines if the binding violates any static sort constraints in the <primary>.

isSatisfyPrimaryCondition$_0$(*bl*: *BINDINGLIST*$_0$, *pr*: <primary>):*BOOLEAN*$=_{def}$
case *pr* **of**
|<operator application>=>*isSatisfyOpAppCondition*$_0$(*bl*, *pr*)
|<method application>=>*isSatisfyMethodAppCondition*$_0$(*bl*, *pr*)
|<expression>=>*isSatisfyExpressionCondition*$_0$(*bl*, *pr*)
|<conditional expression>=>
isSatisfyExpressionCondition$_0$(*bl*, *pr*.**s-**<expression>)$\wedge$
isSatisfyExpressionCondition$_0$(*bl*, *pr*.**s-**<consequence expression>)$\wedge$
isSatisfyExpressionCondition$_0$(*bl*, *pr*.**s-**<alternative expression>)
|<extended primary>=>*isSatisfyExtendedPrimaryCond*$_0$(*bl*, *pr*)
otherwise *True*
endcase

The function *isSatisfyExtendedPrimaryCond*$_0$ determines if the binding violates any static sort constraints in the <extended primary>.

isSatisfyExtendedPrimaryCond$_0$(*bl*: *BINDINGLIST*$_0$, *epr*: <extended primary>):*BOOLEAN*$=_{def}$
case *epr* **of**
|<indexed primary>=>
isSatisfyPrimaryCondition$_0$(*epr*.**s-**<primary>, *bl*)$\wedge$
($\forall i \in$ 1..*epr*.**s-**<actual parameter>-**seq**.*length*:
isSatisfyExpressionCondition$_0$(*bl*,*epr*.**s-**<actual parameter>-**seq**[*i*]))$\wedge$
isSortCompatible$_0$(*getStaticSort*$_0$(*epr*.**s-**<actual parameter>-**seq**[1], *bl*),
getIndexSort$_0$(*getPrimarySort*$_0$(*epr*.**s-**<primary>, *bl*)))
|<field primary>=>
(*epr*.**s-**<primary> $\neq$ *undefined*)$\Rightarrow$
(*isSatisfyPrimaryCondition*$_0$(*epr*.**s-**<primary>, *bl*)$\wedge$
getFieldSort$_0$(*getPrimarySort*$_0$(*epr*.**s-**<primary>, *bl*), *epr*.**s-**<field name>) $\neq$ *undefined*)
|<composite primary>=>
let *sl* = <*getStaticSort*$_0$(*para*, *bl*)| *para* **in** *epr*.**s-**<actual parameter>-**seq**> **in**
epr.**s-**<actual parameter>-**seq**$\neq$*empty*$\Rightarrow$
(*getCompositeSort*$_0$(*sl*)$\neq$ *undefined* $\wedge$
($\forall i \in$ 1..*epr*.**s-**<actual parameter>-**seq**.*length*: *epr*.**s-**<actual parameter>-**seq**[*i*]= *undefined*$\vee$
isSatisfyExpressionCondition$_0$(*bl*,*epr*.**s-**<actual parameter>-**seq**[*i*])))
endlet
endcase

The function *getCompositeSort*$_0$ 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.derivedDateType0.identifier0
    endlet
```

The function *getPrimarySort*$_0$ gets the static sort of an <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>=>"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 *getVarAccessSort*$_0$ 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 *getImperativeExpSort*$_0$ gets the static sort of an <imperative expression>.

```
getImperativeExpSort0(ie: <imperative expression>): <sort>= def
    case ie of
    |<now expression>=> <identifier>(<qualifier>(<name>("Predefined")), "Time")
    |<import expression>=>getEntityDefinition0(ie.s-<identifier>, remote variable).s-<sort>
    |<pid expression>=>
        if ie ≠ self then <identifier>(<qualifier>(<name>("Predefined")), "Pid")
        else
            let def = parentAS0ofKind(ie, <agent definition>∪<agent type definition>) in
            if def = undefined then <identifier>(<qualifier>(<name>("Predefined")), "Pid")
            else def.identifier0
            endif  endlet
        endif
    |<any expression>=>ie.s-<sort>
    |<state expression>=><identifier>(<qualifier>(<name>("Predefined")), "Charstring")
    endcase
```

The function *getSynonymSort*$_0$ gets the static sort of an <synonym>.

getSynonymSort$_0$(*s*: <synonym>): <sort>=$_{\text{def}}$
let *sd* = *getEntityDefinition*$_0$(*pr*, **synonym**) **in**
if *sd*.**s-**<sort> ≠ *undefined* **then** *sd*.**s-**<sort>
else *take*(*sd*.**s-**<constant expression>.*staticSortSet*$_0$)
endif endlet

The function *isSatisfyOpAppCondition*$_0$ determines if the binding violates any static sort constraints in the <operator application>.

isSatisfyOpAppCondition$_0$(*bl*:*BINDINGLIST*$_0$, *oa*:<operation application>): *BOOLEAN*=$_{\text{def}}$
let *opDef* = *getDefinitionInBindingList*$_0$(*oa*.**s-**<operation identifier>.**s-**<name>, *bl*) **in**
let *fpl* = *opDef.operationParameterSortList*$_0$ **in**
let *apl* =*oa.actualParameterList*$_0$ **in**
fpl.length = *apl.length* ∧
(∀*i*∈ 1..*fpl.length*: *isSortCompatible*$_0$(*getStaticSort*$_0$(*apl*[*i*], *bl*), *fpl*[*i*])∧
isSatisfyExpressionCondition$_0$(*bl*, *apl*[*i*])
endlet

The function *getOpAppSort*$_0$ gets the static sort of an <synonym>.

getOpAppSort$_0$(*oa*:<operation application>, *bl*: *BINDINGLIST*$_0$): <sort>=$_{\text{def}}$
getDefinitionInBindingList$_0$(*oa*.**s-**<operation identifier>.**s-**<name>, *bl*).*operationResult*$_0$

The function *isSatisfyMethodAppCondition*$_0$ determines if the binding violates any static sort constraints in the <method application>.

isSatisfyMethodAppCondition$_0$(*bl*:*BINDINGLIST*$_0$, *ma*: <method application>): *BOOLEAN*=$_{\text{def}}$
let *opDef* = *getDefinitionInBindingList*$_0$(*ma*.**s-**<operation identifier>.**s-**<name>, *bl*) **in**
let *fpl* = *opDef.operationParameterSortList*$_0$ **in**
let *apl* =*ma.actualParameterList*$_0$ **in**
fpl.length = *apl.length* ∧*isSatisfyPrimaryCondition*$_0$(*bl*, *ma*.**s-**<primary>)∧
(∀*i*∈ 1..*fpl.length*: *isSortCompatible*$_0$(*getStaticSort*$_0$(*apl*[*i*], *bl*), *fpl*[*i*])∧
isSatisfyExpressionCondition$_0$(*bl*, *apl*[*i*])
endlet

The function *operationParameterSortList*$_0$ gets the operation formal parameter sort list.

operationParameterSortList$_0$(*os*:<operation signature>):<sort>*=$_{\text{def}}$
if *os*. *getPredefinedOpParas*$_0$ ≠ *undefined* **then**
os.getPredefinedOpParas$_0$
else
<*paras*.**s-**<formal parameter>.**s-**<sort> | *paras* **in** *os.operationSignatureParameterList*$_0$>
endif

The function *getDefinitionInBindingList*$_0$ gets the corresponding entity definition for a name in a binding list.

getDefinitionInBindingList$_0$(*n*: <name>,*bl*: *BINDINGLIST*$_0$): *ENTITYDEFINITION*$_0$=$_{\text{def}}$
take({*d*∈ *ENTITYDEFINITION*$_0$: ∃*i*∈ 1..*bl.length*: *bl*[*i*].**s-**<name>=*n*∧ *bl*[*i*].**s-***ENTITYDEFINITION*$_0$=*d*})

The function *possibleDefinitionSet*$_0$ gets the set of possible entity definition for a name.

possibleDefinitionSet$_0$(*n*: <name>):*ENTITYDEFINITION*$_0$**-set** =$_{\text{def}}$
{*d*∈ *ENTITYDEFINITION*$_0$: ((*d.entityName*$_0$=*n*)∨(*isRenamedBy*$_0$(*d.entityName*$_0$, *n*)))∧
((*d.entityKind*$_0$=*n.idKind*$_0$∧*isVisibleIn*$_0$(*d*, *n.surroundingScopeUnit*$_0$))∨
(*d*∈ *PREDEFINEDLITERAL*$_0$ ∧ *n.idKind*$_0$=**literal**)∨
(*d*∈ *PREDEFINEDOPERATION*$_0$∧ *n.idKind*$_0$∈ {**operator**, **method**}))}

The function *isRenamedBy*$_0$ determines if a name is renamed by another one.

isRenamedBy$_0$(*n1*, *n2*: <name>):*BOOLEAN*=$_{\text{def}}$
(∃*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>: *isRenamedBy*$_0$(*n1*, *n3*)∧ *isRenamedBy*$_0$(*n3*, *n2*))

The function *actualParameterList*$_0$ gets the actual parameter list for a <create expression>, a <value returning procedure call> or an <operation application>.

actualParameterList$_0$(*d*: <create expression>∪<value returning procedure call>∪<operation application>): <expression>*=$_{def}$
d. **s-**<actual parameter>**-seq**

The function *getCreatedAgentDefinition*$_0$ gets the <agent definition> or <agent type definition> that the <create expression> involves.

getCreatedAgentDefinition$_0$(*ce*: <create expression>): <agent definition>∪<agent type definition>=$_{def}$
let *id*= *ce*.**s-implicit in**
if *id*∈ <identifier> **then** *getEntityDefinition*$_0$(*id*, *id.idKind*$_0$)
elseif *id.surroundingScopeUnit*$_0$∈ <agent definition>∪<agent type definition> **then**
id.surroundingScopeUnit$_0$
else *undefined*
endif

The function *getStaticSort*$_0$ gets the static sort of an expression.

getStaticSort$_0$(*exp*: <expression>, *bl*:*BINDINGLIST*$_0$): <sort>=$_{def}$
case *exp* **of**
|<create expression>=>*getCreateExpSort*$_0$(*exp*)
|<value returning procedure call>=>
if *exp*∈ <procedure call body> **then** *getProcCallBodySort*$_0$(*exp*)
else *getRemoteProcCallBodySort*$_0$(*exp*, *bl*)
endif
|<range check expression> => Boolean
|<equality expression>=> Boolean
|<binary expression>=> *getDefinitionInBindingList*$_0$(*exp*.**s-implicit**, *bl*).*operationResult*$_0$
|<expression gen primary>=>
if *exp*.**s-implicit** = *undefined* **then**
getPrimarySort$_0$(*pr*, *bl*)
else
let *opDef* = *getDefinitionInBindingList*$_0$(*exp*.**s-implicit**, *bl*) **in**
opDef.operationResult$_0$
endlet
endif
endcase

The function *mismatchNumber*$_0$ 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.

mismatchNumber$_0$(*bl*:*BINDINGLIST*$_0$, *c*:*CONTEXT*$_0$): *NAT*=$_{def}$
case *c* **of**
|<assignment>=>*mismatchNumberOfAssignment*$_0$(*bl*, *c*)
|<decision>=>*mismatchNumberOfDecision*$_0$(*bl*, *c*)
|<expression>=>*mismatchNumberOfExpression*$_0$(*bl*, *c*)
endcase

mismatchNumberOfAssignment$_0$(*bl*:*BINDINGLIST*$_0$, *ass*: <assignment>): *NAT*=$_{def}$
let *varSort*= *getVariableSort*$_0$(*ass*.**s-**<variable>) **in**
let *expSort*= *getStaticSort*$_0$(*ass*.**s-**<expression>, *bl*) **in**
case *ass*.**s-**<variable> **of**
| <identifier>∪<field variable>=>

if ¬*isSameSort*$_0$(*varSort*, *expSort*) **then** 1+*mismatchNumberOfExpression*$_0$(*bl*, *ass*.**s-**<expression>)
else *mismatchNumberOfExpression*$_0$(*bl*, *ass*.**s-**<expression>)
endif
|<indexed variable>=>
if ¬*isSameSort*$_0$(*varSort*, *expSort*) **then**
1 + *mismatchNumberOfExpression*$_0$(*bl*, *ass*.**s-**<expression>)+
mismatchNumberInIndexVariable$_0$(*bl*, *ass*.**s-**<variable>)
else
mismatchNumberOfExpression$_0$(*bl*, *ass*.**s-**<expression>)+
mismatchNumberInIndexVariable$_0$(*bl*, *ass*.**s-**<variable>)
endif
endcase

mismatchNumberInIndexVariable$_0$(*bl*:*BINDINGLIST*$_0$, *var*:<indexed variable>): *NAT*=$_{\text{def}}$
let *acp*=*var*.**s-**<actual parameter>**-seq in**
if *var*.**s-**<variable>∉ <indexed variable> **then**
if ¬*isSameSort*$_0$(*getStaticSort*$_0$(*acp*[1],*bl*), *getIndexSort*$_0$(*getVariableSort*$_0$(*var*))) **then** 1
else 0
endif
elseif ¬*isSameSort*$_0$(*getStaticSort*$_0$(*acp*[1],*bl*), *getIndexSort*$_0$(*getVariableSort*$_0$(*var*))) **then**
1 + *mismatchNumberInIndexVariable*$_0$(*bl*, *var*.**s-**<variable>)
else *mismatchNumberInIndexVariable*$_0$(*bl*, *var*.**s-**<variable>)
endif
endlet

mismatchNumberOfDecision$_0$(*bl*:*BINDINGLIST*$_0$, *d*: <decision>): *NAT*=$_{\text{def}}$
let *q*=*d*.**s-**<question>, *rcl*=*d.rangeConditionList*$_0$ **in**
mismatchNumberOfExpression$_0$(*bl*, *q*) + *mismatchNumberOfRangeCondtionList*$_0$(*bl*, *rcl*))
endlet

mismatchNumberOfRangeCondtionList$_0$(*bl*:*BINDINGLIST*$_0$, *rcl*: <range condition>*): *NAT*=$_{\text{def}}$
if *rcl*.= *empty* **then** *empty*
else
mismatchNumberOfRangeCond$_0$(*bl*, *rcl.head*)+ *mismatchNumberOfRangeCondtionList*$_0$ (*bl*, *rcl.tail*])
endif

mismatchNumberOfRangeCond$_0$(*bl*:*BINDINGLIST*$_0$, *rl*: <range>*): *NAT*=$_{\text{def}}$
if *rl* = *empty* **then** *empty*
else
mismatchNumberOfRange$_0$(*bl*, *rl.head*)+ *mismatchNumberOfRangeCond*$_0$ (*bl*, *rl.tail*])
endif

mismatchNumberOfRange$_0$(*bl*:*BINDINGLIST*$_0$, *range*: <range>): *NAT*=$_{\text{def}}$
case *range* **of**
|<closed range>=>
mismatchNumberOfExpression$_0$(*bl*,*range*.**s-**<constant>) +
mismatchNumberOfExpression$_0$(*bl*,*range*.**s2-**<constant>)
|<open range>=>
if *range*∈ <constant> **then** *mismatchNumberOfExpression*$_0$(*bl*,*range*)
else
mismatchNumberOfExpression$_0$(*bl*,*range*.**s-**<constant>)+
mismatchNumberOfExpression$_0$(*bl*,*range*.**s2-**<constant>)
endif
endcase

mismatchNumberOfExpression$_0$(*bl*:*BINDINGLIST*$_0$, *exp*: <expression>): *NAT*=$_{\text{def}}$
case *exp* **of**
|<create expression>=>*mismatchNumberOfCreateExp*$_0$(*bl*, *exp*)
|<value returning procedure call>=>
if *exp*∈ <procedure call body> **then**
mismatchNumberOfProcedureCallBody$_0$(*bl*, *call*)
else
mismatchNumberOfRemoteProcCallBody$_0$(*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
    |<expression gen primary>=>
        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
    | <identifier> =>
        let fpsl=getEntityDefinition0(id, procedure).formalParameterSortList0 in
            mismatchNumberOfParas0(bl, fpsl, apl)
        endlet
    | <type expression>=>
        let fpsl=id.baseType0.formalParameterSortList0 in
            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

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
```

mismatchNumberOfOpApp$_0$(*bl*:*BINDINGLIST*$_0$, *oa*: <operator application>): *NAT*=$_{\text{def}}$
let *opDef* = *getDefinitionInBindingList*$_0$(*oa*.**s-**<operation identifier>.**s-**<name>, *bl*) **in**
let *fpsl* = *opDef.operationParameterSortList*$_0$ **in**
let *apl* =*oa.actualParameterList*$_0$ **in**
mismatchNumberOfParas$_0$(*bl*, *fpsl*, *apl*)+*mismatchNumberOfActualParas*$_0$(*bl*, *apl*)
endlet

mismatchNumberOfMethodApp$_0$(*bl*:*BINDINGLIST*$_0$, *ma*: <method application>):*NAT*=$_{\text{def}}$
let *opDef* = *getDefinitionInBindingList*$_0$(*ma*.**s-**<operation identifier>.**s-**<name>, *bl*) **in**
let *fpsl* = *opDef.operationParameterSortList*$_0$ **in**
let *apl* =*ma.actualParameterList*$_0$ **in**
mismatchNumberOfParas$_0$(*bl*, *fpsl*, *apl*)+*mismatchNumberOfActualParas*$_0$(*bl*, *apl*)+
mismatchNumberOfPrimary$_0$(*bl*, *ma*.**s-**<primary>)
endlet

mismatchNumberOfParas$_0$(*bl*: *BINDINGLIST*$_0$, *fpsl*: <sort>*, *apl*: <expression>*):*NAT*=$_{\text{def}}$
if *fpsl* = *empty* **then** 0
elseif *fpsl.head*=*undefined*∨*isSameSort*$_0$(*fpsl.head*, *apl.head*) **then** *mismatchNumberOfParas*$_0$(*bl*, *fpsl.tail*, *apl.tail*)
else *mismatchNumberOfParas*$_0$(*bl*, *fpsl.tail*, *apl.tail*)+1
endif

mismatchNumberOfActualParas$_0$(*bl*: *BINDINGLIST*$_0$, *apl*: <expression>*): *NAT*=$_{\text{def}}$
if *apl* = *empty* **then** 0
else *mismatchNumberOfExpression*$_0$(*bl*, *apl.head*)+*mismatchNumberOfActualParas*$_0$(*bl*, *apl.tail*)
endif

formalParameterSortList$_0$(*d*: <agent definition>∪<agent type definition>∪<composite state>∪<composite state type definition>∪<procedure definitions>∪<remote procedure definition>):<sort>*=$_{\text{def}}$
case *d* **of**
|<agent definition>∪<agent type definition>∪<composite state>∪<composite state type definition>=>
<*fp.parentAS1*.**s-**<sort> | *fp* **in** *d.agentFormalParameterList*$_0$>
|<procedure definitions>=>
<*fp.parentAS1*.**s-**<sort> | *fp* **in** *d.procedureFormalParameterList*$_0$>
|<remote procedure definition>=>
<*fp*.**s-**<sort> | *fp* **in** *d*.**s-**<procedure signature>.**s-**<formal parameter>>
endcase

The function *staticSortSet*$_0$ gets the possible static sort set for an <expression>.

staticSortSet$_0$(*exp*: <expression>):<sort>-**set** =$_{\text{def}}$
{*getStaticSort*$_0$(*exp*, *bl*): *bl*∈*getBindingListSet*$_0$(*exp.contextOfExp*$_0$)}

The function *contextOfExp*$_0$ finds the context that the <expression> is in.

contextOfExp$_0$(*exp*: <expression>):*CONTEXT*$_0$=$_{\text{def}}$
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>).*contextOfExp*$_0$
endif

The function *getPredefinedOpParas*$_0$ gets the sort list of the formal parameters of a predefined operation. This function is not formally defined in this Recommendation.

getPredefinedOpParas$_0$: *PREDEFINEDOPERATION*$_0$ → <sort>*

The function *getPredefinedOpResult*$_0$ gets the result sort of a predefined operation. This function is not formally defined in this Recommendation.

getPredefinedOpResult$_0$: *PREDEFINEDOPERATION*$_0$ → <sort>

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*
	|	*Signal-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*
Signal-qualifier	::	*Signal-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>(**SIGNAL**,*n*)	=> **mk-***Signal-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*)

2.3 Informal Text

Abstract Syntax

Informal-text :: ...

Concrete Syntax

<informal text> :: <character string>

Mapping to Abstract Syntax

The mapping for informal text is already described within 2.2.1.

2.4 General Framework

2.4.1 SDL 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>
| <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.name*$_0$, <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*))

2.4.2 Package

Abstract Syntax

Package-definition :: *Package-name*
Package-definition-**set**
Data-type-definition-**set**

Syntype-definition-**set**
Signal-definition-**set**
Exception-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 interface>

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

<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**
  | **exception**
  | **interface**

<package interface> = <definition selection>*

Conditions on Concrete Syntax

$\forall id \in$ <identifier>:
($id.parentAS0 \in$ <package use clause>)$\Rightarrow$*getEntityDefinition*$_0$(*id*, **package**) $\neq$ *undefined*

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

$\forall pd \in$ <package definition>:
$pd.parentAS0 \in$ <sdl specification> $\Rightarrow pd.qualifier_0 = undefined$

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

$\forall ds1, ds2 \in$ <definition selection>:
$(ds1.parentAS0 = ds2.parentAS0 \wedge ds1 \neq ds2) \Rightarrow$
($ds1$.**s-**<selected entity kind> $\neq ds2$.**s-**<selected entity kind> $\vee$ $ds1$.**s-**<name> $\neq ds2$.**s-**<name>)

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

$\forall ds \in$ <definition selection>:
ds.**s-**<selected entity kind> $= undefined \wedge d.parentAS0 \in$ <package interface> $\Rightarrow$
($\exists!\ d \in ENTITYDEFINITION_0 \cup REFERENCE_0$:
$d.surroundingScopeUnit_0 = ds.parentAS0.parentAS0 \wedge d.entityName_0 =$ ds.**s-**<name>)

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

$\forall\ uc \in$ <package use clause>:$\forall ds \in$ <definition selection>:
let $pd = uc.usedPackage_0$ **in**
$ds.parentAS0 = uc \wedge ds$.**s-**<selected entity kind> $= undefined \Rightarrow$
(($\exists!\ ds1 \in$ <definition selection>: $ds1.surroundingScopeUnit_0 = pd \wedge ds$.**s-**<name>= $ds1$.**s-**<name>) $\vee$
((pd. **s-**<package heading>.**s-**<package interface> $= undefined$)$\wedge$
($\exists!\ d \in ENTITYDEFINITION_0 \cup REFERENCE_0$:
$d.surroundingScopeUnit_0 = pd \wedge d.entityName_0 =$ 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.

$\forall ds \in$ <definition selection>:
ds.**s-**<selected entity kind> $\neq undefined \wedge d.parentAS0 \in$ <package interface> $\Rightarrow$
($\exists\ d \in ENTITYDEFINITION_0 \cup REFERENCE_0$:
$d.surroundingScopeUnit_0 = ds.surroundingScopeUnit_0 \wedge d.entityName_0 = ds$.**s-**<name> $\wedge$
$d.entityKind_0 = ds$.**s-**<selected entity kind>)

For each pair of (<selected entity kind>, <name>) in <package interface>, 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>(*undefined*,<name>("Predefined")), *undefined*) **in**
<package definition>(*uses*, *heading*, *entities*) **provided** *uses.head* ≠ *usePredef* =6=>
<package definition>(< *usePredef* > ^ *uses*, *heading*, *entities*)

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

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

use Predefined;

where Predefined denotes a package containing the predefined data as defined in Annex D.

< <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> which 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* ∈ *Exception-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 *usedPackageDefinitionList*$_0$ gets the used package definition list of a scope unit.

usedPackageDefinitionList$_0$(*su*: *SCOPEUNIT*$_0$): <package definition> *=$_{\text{def}}$
< *u*.*usedPackage*$_0$ | *u* **in** *su*.**s-**<package use clause> >

The function *usedPackage*$_0$ gets the package definition for a <package use clause>.

usedPackage$_0$(*uc*: <package use clause>): <package definition> =$_{\text{def}}$
getEntityDefinition$_0$(*uc*.**s-**<identifier>, **package**)

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 definitions>
| <operation definition>

<package reference> :: <package<identifier>

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

<procedure reference> ::
<type preamble> <procedure<identifier>

<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>
<block type reference>:: <type preamble> <block type<identifier>
<process type reference> :: <type preamble> <process type<identifier>

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

<signal reference> :: <type preamble> <signal<identifier>

<data type reference> :: <type preamble> <data type kind> <data type<identifier>

<interface reference> :: [<virtuality>] <interface<identifier>

Conditions on Concrete Syntax

$\forall refDef \in$ <referenced definition>: $refDef.referencedBy_0 \neq undefined$

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

$\forall ref \in REFERENCE_0 : ref.referencedDefinition_0 \neq undefined$

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.

$\forall ref1, ref2 \in$ <referenced definition>:
$ref1.entityName_0 = ref2.entityName_0 \wedge ref1.entityKind_0 = ref2.entityKind_0 \wedge ref1 \neq ref2 \Rightarrow$
$ref1.qualifier_0 \neq undefined \wedge ref2.qualifier_0 \neq undefined \wedge$
$\neg isPathItemMatched_0$ ($ref1.qualifier_0$.**s-**<path item>**-seq** , $ref2.qualifier_0$.**s-**<path item>**-seq**) $\wedge$
$\neg isPathItemMatched_0$ ($ref2.qualifier_0$.**s-**<path item>**-seq** , $ref1.qualifier_0$.**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.

$\forall rd \in$ <referenced definition>:
$rd \in$ <package definition> $\Rightarrow rd.qualifier_0 \neq undefined$

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

$\forall def \in ENTITYDEFINITION_0$: $def \notin$ <referenced definition> $\Rightarrow def.qualifier_0 = undefined$

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.referencedDefinition*$_0$, *undefined*)

p= <procedure reference>(*,*) =4=> *adaptDefinition*(*p.referencedDefinition*$_0$, *undefined*)

b= <block reference>(*, *inst*) =4=> *adaptDefinition*(*b.referencedDefinition*$_0$, *inst*)

p= <process reference>(*, *inst*) =4=> *adaptDefinition*(*p.referencedDefinition*$_0$, *inst*)

i= <interface reference>(*,*) =4=> *adaptDefinition*(*i.referencedDefinition*$_0$, *undefined*)

c= <composite state reference>(*) =4=> *adaptDefinition*(*c.referencedDefinition*$_0$, *undefined*)

s= <system type reference>(*) =4=> *adaptDefinition*(*s.referencedDefinition*$_0$, *undefined*)

b= <block type reference>(*,*) =4=> *adaptDefinition*(*b.referencedDefinition*$_0$, *undefined*)

p= <process type reference>(*,*) =4=> *adaptDefinition*(*p.referencedDefinition*$_0$, *undefined*)

c= <composite state type reference>(*,*) =4=> *adaptDefinition*(*c.referencedDefinition*$_0$, *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 *referencedDefinition*$_0$ finds the corresponding entity definition for a given reference.

referencedDefinition$_0$(*ref*: *REFERENCE*$_0$): <referenced definition> =$_{def}$
let *eKind* = *ref.referenceKind*$_0$ **in**
let *refName* = *ref.referenceName*$_0$ **in**
if (∃! *d*∈ <referenced definition>:
isAncestorAS0(*d.parentAS0*, *ref*) ∧*d.entityName*$_0$ = *refName* ∧ *d*. *entityKind*$_0$ =*eKind*) **then**
let *d* = *take*({*d*∈ <referenced definition>:
isAncestorAS0(*d.parentAS0*, *ref*) ∧ *d.entityName*$_0$ = *refName* ∧ *d*. *entityKind*$_0$ =*eKind*}) **in**
if *isQualifierMatched*$_0$(*d.qualifier*$_0$, *ref.surroundingScopeUnit*$_0$) **then** *d*
else
undefined
endif
else
undefined
endif
endlet

The function *referencedBy*$_0$ finds the corresponding reference for a <referenced definition>.

referencedBy$_0$(*rd*: <referenced definition>): *REFERENCE*$_0$ =$_{def}$
take({ *ref*∈ *REFERENCE*$_0$: *ref.referencedDefinition*$_0$ = *rd*})

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

isPathItemMatched$_0$(*seq1*: <path item>*, *seq2*: <path item>*): *BOOLEAN* =$_{def}$
(*seq1.length*≤ *seq2.length*∧
(∀*i*∈ 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*)
| <procedure definition>(*uses*,

<procedure heading>(*h1*, *, *name*, *h2*, *h3*, *h4*, *h5*, *h6*, *h7*),
entities, *body*)
=> <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

2.4.4 Select Definition

Concrete Syntax

<select definition> ::
<Boolean><simple expression>
{ <agent type definition>
| <agent type reference>
| <agent definition>
| <channel definition>
| <signal definition>
| <signal list definition>
| <remote variable definition>
| <remote procedure definition>
| <data definition>
| <data type reference>
| <composite state type definition>
| <composite state type reference>
| <state partition>
| <timer definition>
| <variable definition>
| <procedure definitions>
| <procedure reference>

| <channel to channel connection>
| <select definition>
| <exception definition>}+

Transformations

< <select definition>(*cond*, *defs*) >
=7=> **if** $value_0$(*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. Any connectors connected to an area within non-selected <option area>s are removed too.

2.4.5 Transition Option

Concrete Syntax

<transition option> :: <alternative question> <decision body>
<alternative question> = <simple expression> | <informal text>

Conditions on Concrete Syntax

$\forall ce \in$ <constant expression>:*parentAS0ofKind*(*ce*, <transition option>)$\neq$*undefined*$\Rightarrow$$isSimpleExpression_0$(*ce*)

Every <constant expression> in <answer> of <decision body> must be a <simple expression>.

$\forall to \in$ <transition option>: *to*.**s-**<alternative question>$\in$ <simple expression>$\Rightarrow$
($\forall rc \in$ <range condition>: *rc* **in** *to*.**s-**<decision body>.**s-**<answer part>.$answerPartRangeConditionList_0\Rightarrow$
($\forall ce \in$ <constant expression>: *isAncestorAS0*(*rc*, *ce*)$\Rightarrow$
$isSameSort_0$(*to*.**s-**<alternative question>.$staticSort_0$, *ce*.$staticSort_0$)))

If the <alternative question> is an <expression>, the <range condition> of the <answer>s in the <decision body> must be of the same sort as of the <alternative question>.

Transformations

t=<transition gen action statement>(*a*, *undefined*)
provided *a.last* $\notin$ <transition option> $\wedge$ *t.parentAS0.parentAS0.parentAS0* $\in$ <transition option> $\wedge$
t.findContinueLabel $\neq$ *undefined*
=7=> <transition gen action statement>(*a*,
<terminator statement>(*undefined*,<join>(*findContinueLabel*(*t*))))

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

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

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

<transition option>(*q*, <decision body>(*answers*, *elsePart*))
=7=> (**let** *matching* = { *a*.**s-**<transition> | *a* $\in$ *answers.toSet*: *q*. $value_0 \in$ *a*.**s-**<answer> } **in**
if *matching* $\neq \emptyset$ **then** *matching.take*
elseif *elsePart* $\neq$ *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.

2.4.6 Associations

Associations do not have a semantics in SDL.

2.5 Structural Concepts

2.5.1 Structural Type Definitions

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** *Exception-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-definition*]
Agent-kind	=	**SYSTEM** | **BLOCK** | **PROCESS**

Conditions on Abstract Syntax

$\forall d \in$ *Agent-type-definition*: $d.agentKind_1 \in$ {**system**, **process**} $\Rightarrow$
(d.**s-***Agent-definition*-**set** $\neq \varnothing \vee$ d.**s-***State-machine-definition* $\neq$ *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-definition*.

$\forall d \in$ *Agent-type-definition*: ($d.agentKind_1$ = **process**) $\Rightarrow$
($\forall$ *d'* $\in$ *Agent-type-definition*$\cup$*Agent-definition*: *d'.parentAS1*= $d \Rightarrow$ *d'*. $agentKind_1$= **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 parameter>* [<virtuality constraint>] <agent additional heading>

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

Transformations

a = <agent body>(*excAndStart*, *items*) =1=>
  <interaction>(*empty*,
    <composite state>(*empty*, <composite state heading>(*empty*, *a.parentAS0.parentAS0.*$name_0$, *empty*),
      <composite state structure>(*undefined*, *empty*, *empty*, *empty*,
        <composite state body>(

if *excAndStart* = *undefined* **then** *undefined* **else** *excAndStart*.**s-**<on exception> **endif**,
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

Gets all the <procedure definitions> defined locally in an <agent type definition>.

agentTypeLocalProcedureDefinitionSet$_0$(*td*: <agent type definition>):<procedure definition>-**set** =$_{def}$
{*pd*∈ <procedure definition>: *pd.surroundingScopeUnit*$_0$= *td*}

Gets the set of explicit defined output signals for an <agent type definition> or an <agent definition>.

validOutputSignalSet$_0$(*d*: <agent type definition>∪<agent definition>):*SIGNAL*$_0$ =$_{def}$
d.localOutputSignalSet$_0$∪ *d.inheritedOutputSignalSet*$_0$

localOutputSignalSet$_0$(*d*: <agent type definition>∪<agent definition>):*SIGNAL*$_0$ =$_{def}$
{*sid*∈ *SIGNAL*$_0$:
(∃*gc*∈ <gate constraint>:
(*gc.parentAS0*∈ <textual gate definition>)∧*isDefinedIn*$_0$(*gc.parentAS0*, *d*)∧
(*gc.direction*$_0$=**out**)∧(*sid*∈ *signalSet*$_0$(*gc*.**s-**<signal list item>**-seq**)) ∨
(∃*cp*∈ <channel path>:
(*cp.parentAS0*∈ <channel definition>)∧*isDefinedIn*$_0$(*cp.parentAS0*, *d*)∧
(*cp.destination*$_0$.**s-env**≠*undefined*)∧(*sid*∈ *signalSet*$_0$(*cp*.**s-**<signal list item>**-seq**)))}

inheritedOutputSignalSet$_0$(*d*: <agent type definition>∪<agent definition>):*SIGNAL*$_0$ =$_{def}$
if *d.specialization*$_0$ = *undefined* **then** ∅
else *d.specialization*$_0$.**s-**<type expression>.*baseType*$_0$.*validOutputSignalSet*$_0$
endif

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.surroundingScopeUnit*$_0$∈ <system type definition>) ⇒
(*fcp*∉ <agent context parameter> ∪ <variable context parameter> ∪ <timer context parameter>)

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

¬(∃*fps*∈ <formal agent parameter>: *fps.surroundingScopeUnit*$_0$∈ <system type definition>)

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

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* ∈ *Exception-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*))

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* ∈ *Exception-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*))

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* ∈ *Exception-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*))

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**
Exception-definition-**set**
Composite-state-type-definition-**set**
Variable-definition-**set**
Procedure-definition-**set**
[*Composite-state-graph* | *State-aggregation-node*]

Conditions on Abstract Syntax

∀*d*∈ *Composite-state-type-definition*: *d*.**s-***Gate-definition*-**set** ≠ ∅⇒
(∃*sd*∈ *State-machine-definition*:*getEntityDefinition*$_1$(*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-definition*.

Concrete Syntax

<composite state type definition> ::
<package use clause>* <composite state type heading> <composite state structure>

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

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* ∈ *Exception-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* ∈ *Exception-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 gen entry> ∧ *c*.**s-**<entry point>.**s-**<identifier>=*b.identifier*$_0$)
> ∩
< *Mapping*(*c*) | *c* **in** *body*:
(*c* ∈ <state partition connection gen exit> ∧ *c*.**s-**<exit point>.**s-**<identifier> = *b.identifier*$_0$)
>) |
b **in** *body*: (*b* ∈ <textual typebased state partition definition>) >,
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")>)))

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* ⇒ *isParameterizedType*$_0$(*te.baseType*$_0$)

<actual context parameters> can be specified 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.refersto*$_0$ >
=> < *id.refersto*$_0$,
replaceContextParameters$_0$(*id.refersto*$_0$*.localFormalContextParameterList*$_0$, *params*,
createNewType$_0$(*nn*, *id.refersto*$_0$) >

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 $isDirectSubType_0$ determines if a type definition is a direct subtype of an entity definition.

$isDirectSubType_0$(*ed*: $ENTITYDEFINITION_0$, *td*: $TYPEDEFINITION_0$): $BOOLEAN$=$_{def}$
∃*te*∈ <type expression>: *te*. *parentAS0*=*ed*. $specialization_0$∧*td*=*te*.$baseType_0$

The function $isSubtype_0$ determines if a type definition is a subtype of an entity definition.

$isSubtype_0$(*sub*: $ENTITYDEFINITION_0$, *sup*: $TYPEDEFINITION_0$): $BOOLEAN$=$_{def}$
$isDirectSubType_0$(*sub*, *sup*) ∨
(∃*ttd*∈ $TYPEDEFINITION_0$: $isSubtype_0$(*sub*, *ttd*) ∧ $isSubtype_0$(*ttd*, *sup*))

The function $baseType_0$ is used to get the base type definition for a type expression.

$baseType_0$(*te*: <type expression>): $TYPEDEFINITION_0$ =$_{def}$
$getEntityDefinition_0$(*te*.**s**-<base type>, *te*.$baseTypeKind_0$)

The function $baseTypeKind_0$ is used to get the base type kind for a type expression.

$baseTypeKind_0$(*te*: <type expression>): $ENTITYKIND_0$=$_{def}$
case *te.parentAS0* **of**
| <specialization>∪<data type specialization>∪<interface specialization> =>
if (*te*.$surroundingScopeUnit_0$∈ <agent definition>) **then agent type**
else *te*.$surroundingScopeUnit_0$.$entityKind_0$
| <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 $isParameterizedType_0$ is used to determine if a type definition is a parameterized type.

$isParameterizedType_0$(*td*: $TYPEDEFINITION_0$): $BOOLEAN$=$_{def}$
(*td*.$formalContextParameterList_0$= *empty*)

Get the formal context parameter list of a type definition.

$formalContextParameterList_0$(*td*: $TYPEDEFINITION_0$): <name>* =$_{def}$
td.$inheritedFormalContextParameterList_0$^ *td*.$localFormalContextParameterList_0$

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

$inheritedFormalContextParameterList_0$(*td*: $TYPEDEFINITION_0$): <name>* =$_{def}$
let *sp*=*td*.$specialization_0$ **in**
if *sp* = *undefined* **then** *empty* **else**
case *sp* **of**
| <interface specialization> =>
< $getUnboundFormalContextParameterList_0$(*tel*.$baseType_0$.$formalContextParameterList_0$,
tel.$actualContextParameterList_0$)
| *tel* **in** *sp*.**s**-<type expression>-**seq** >
otherwise
let *fcpl*=*sp*.**s**-<type expression>.$baseType_0$.$formalContextParameterList_0$**in**
let *acpl*=*sp*.**s**-<type expression>.$actualContextParameterList_0$ **in**
$getUnboundFormalContextParameterList_0$(*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.

getUnboundFormalContextParameterList$_0$(*fcpl*: <name>*, *acpl*: <identifier>*):<name>* =$_{def}$
if (*fcpl* = *empty*) **then** *empty*
elseif (*acpl.head*= *undefined*) **then**
<*fcpl.head*>^*getUnboundFormalContextParameterList*$_0$(*fcpl.tail*, *acpl.tail*)
else
getUnboundFormalContextParameterList$_0$(*fcpl.tail*, *acpl.tail*)
endif

Insert the original context parameter for the unbound ones.

completeFormalContextParameter$_0$(*fcpl*: <name>*, *acpl*: <identifier>*):<name>* =$_{def}$
if (*fcpl* = *empty*) **then** *empty*
elseif (*acpl.head*= *undefined*) **then**
<*fcpl.head*>^*completeFormalContextParameter*$_0$(*fcpl.tail*, *acpl.tail*)
else
<*acpl.head*>^*completeFormalContextParameter*$_0$(*fcpl.tail*, *acpl.tail*)
endif

Get the actual context parameter list of a type expression.

actualContextParameterList$_0$(*te*: <type expression>):<identifier>* =$_{def}$
te.**s**-<actual context parameter>**-seq**

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

localFormalContextParameterList$_0$(*td*: *TYPEDEFINITION*$_0$): <name>* =$_{def}$
let *fcps* =*take*({*fcps*∈ <agent type additional heading> ∪ <composite state type heading> ∪
<procedure heading> ∪ <signal definition item> ∪ <data type heading> ∪
<interface heading>: *fcps.surroundingScopeUnit*$_0$ = *td*}) **in**
< *fcpl.formalContextParameterSublist*$_0$ | *fcpl* **in** *fcps*.**s**-<formal context parameter>**-seq** >
endlet

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

formalContextParameterSublist$_0$(*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> =>
< *scpl*.**s**-<name> | *scpl* **in** *fcp*.**s**-<signal context parameter gen name>**-seq** >
| <variable context parameter> =>
< *vcpl*.**s**-<name> | *scpl* **in** *fcp*.**s**-<variable context parameter gen name>**-seq** >
| <remote variable context parameter> =>
< *vcpl*.**s**-<name> | *scpl* **in** *fcp*.**s**-<remote variable context parameter gen name>**-seq**>
| <timer context parameter> =>
< *vcpl*.**s**-<name> | *scpl* **in** *fcp*.**s**-<timer context parameter gen name> **-seq**>
| <synonym context parameter> =>
< *vcpl*.**s**-<name> | *scpl* **in** *fcp*.**s**-<synonym context parameter>**-seq**>
| <sort context parameter> => <*fcp*.**s**-<name>>
| <exception context parameter> =>
< *vcpl*.**s**-<name> | *scpl* **in** *fcp*.**s**-<exception context parameter gen name>**-seq**>
| <composite state type context parameter> => <*fcp*.**s**-<name>>
| <gate context parameter> => <*fcp*.**s**-<gate>.**s**-<name>>

| <interface context parameter> =>
< *vcpl*.**s**-<name> | *scpl* **in** *fcp*.**s**-<interface context parameter gen name>**-seq**>
otherwise *empty*
endcase

Replace the context parameters by their values.

replaceContextParameters$_0$(*p*: *DefinitionAS0**, *v*: *DefinitionAS0**, *orig*: *DefinitionAS0*): *DefinitionAS0* =$_{\text{def}}$
if *p* = *empty* **then** *orig*
else *replaceContextParameters*$_0$(*p.tail*, *v.tail*, *replaceInSyntaxTree*(*p.head*, *v.head*, *orig*))
endif

Create a new type without any formal context parameters.

createNewType$_0$(*n*: <name>, *orig*: *DefinitionAS0*): *DefinitionAS0* =$_{\text{def}}$
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*)
| <procedure definition>(*use*, <procedure heading>(*pre*, *q*, *, *, *c*, *spec*, *par*, *res*, *raise*), *ent*, *body*)
=> <procedure definition>(*use*,
<procedure heading>(*pre*, *q*, *n*, *empty*, *c*, *spec*, *par*, *res*, *raise*), *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 item>(*, *, *v*, *spec*, *l*)
=> <signal definition item>(*n*, *empty*, *v*, *spec*, *l*)
otherwise *undefined*
endcase

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

$\forall ad \in$ <textual typebased agent definition> : $\forall te \in$ <type expression>:
(*te.parentAS0.parentAS0* = *ad*) $\Rightarrow$
($\exists s \in$ <start>: ($s \in te.baseType_0.startSet_0$)$\wedge$($s$.**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.

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*))

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*))

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

2.5.3.4 Composite State Definition Based on Composite State Type

Concrete Syntax

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

<textual typebased state partition definition> =
    <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.

2.5.4 Abstract Type

Concrete Syntax

<abstract> :: ()

Conditions on Concrete Syntax

$\forall pd \in$ <procedure definitions>: $isAbstractType_0(pd) \Rightarrow$
$\neg(\exists pc \in$ <procedure call>: $pd = pc.calledProcedure_0)$

An abstract procedure cannot be called.

$\forall ad \in$ <textual typebased agent definition> : $\forall te \in$ <type expression>:
$te.parentAS0.parentAS0 = ad \Rightarrow \neg isAbstractType_0(te.baseType_0)$

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

$\forall td \in TYPEDEFINITION_0$: $isAbstractType_0(td) \Rightarrow$
$\neg(\exists d \in$ <textual typebased agent definition> $\cup$ <textual typebased state partition definition> $\cup$
<typebased composite state>:$\exists te \in$ <type expression>:
$((te.parentAS0.parentAS0 = d) \vee (te.parentAS0 = d)) \wedge (te.baseType_0 = td))$

An abstract type can not be instantiated.

Auxiliary Functions

Determine if a type definition is abstract.

$isAbstractType_0(td: TYPEDEFINITION_0): BOOLEAN =_{def}$
$\exists ab \in$ <abstract>: $ab.surroundingScopeUnit_0 = td$

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

calledProcedure$_0$(*pc*: <procedure call>$\cup$<value returning procedure call>): <procedure definition>$=_{def}$
case *pc* **of**
| <procedure call>=>
let *t* = *pc*.**s-**<procedure call body>.**s-implicit in**
if *t*$\in$ <identifier> **then** *getEntityDefinition*$_0$(*t*, **procedure**)
else *t.baseType*$_0$ // *t*$\in$ <type expression>
| <procedure call body>=>
let *t* = *pc*.**s-implicit in**
if *t*$\in$ <identifier> **then** *getEntityDefinition*$_0$(*t*, **procedure**)
else *t.baseType*$_0$ // *t*$\in$ <type expression>
| <remote procedure call body>=> *getEntityDefinition*$_0$(*pc*, **remote procedure**)
| **otherwise** => *undefined*
endcase

2.5.5 Type Reference

Type references do not have a semantics in SDL.

2.5.6 Gate

Abstract Syntax

Gate-definition	::	*Gate-name*
		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> <gate constraint> [<gate constraint>]

<textual interface gate definition> :: { **out** | **in** } <identifier>

<gate> = <gate<name>

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

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

Conditions on Concrete Syntax

$\forall te \in$ <type expression>:
($te.parentAS0 \in$ <typebased composite state>) $\vee$
($te.parentAS0.parentAS0 \in$ <textual typebased agent definition>$\cup$
<textual typebased state partition definition>) $\Rightarrow$
($\forall gc \in$ <gate constraint> :
($gc.parentAS0 \in$ <textual gate definition>)$\wedge isDefinedIn_0(gc.parentAS0, te.baseType_0) \Rightarrow$
gc.**s**-<signal list item>**-seq** $\neq$ *empty*)

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

$\forall gd \in$ <textual gate definition>: $\forall gc \in$ <gate constraint>:
($gc.parentAS0 = gd$) $\Rightarrow$
(**let** $td = gd.surroundingScopeUnit_0$ **in**
($td.entityKind_0$=**state type**$\Rightarrow$
($\exists td' \in TYPEDEFINITION_0: td' = getEntityDefinition_0(gc$.**s**-<textual endpoint constraint>,**state type**)))$\wedge$
($isConsistentKindTo_0(td.entityKind_0$, **agent type**)$\Rightarrow$
($\exists td' \in TYPEDEFINITION_0: td' = getEntityDefinition_0(gc$.**s**-<textual endpoint constraint>,**agent type**)))
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.

$\forall gc \in$ <gate constraint>:$\forall ce \in$ <channel endpoint>:$\forall ce' \in$ <channel endpoint>:
($gc.parentAS0$.**s**-<gate> = ce.**s**-<gate>) $\wedge$
($ce \neq ce'$) $\wedge$ ($ce.parentAS0 = ce'.parentAS0$) $\wedge$ ($ce.parentAS0 \in$ <channel path>) $\wedge$
($gc.parentAS0 \in$ <textual gate definition>) $\Rightarrow$
(**let** $td = getEntityDefinition_0(gc$.**s**-<textual endpoint constraint>,
$gc.surroundingScopeUnit_0.entityKind_0$) **in**
$\exists td' \in ENTITYDEFINITION_0$: ($td' = ce'.channelEndpointReferTo_0$) $\wedge$
(($td = td'$) $\vee$ $isSubtype_0(td', td)$) **endlet**) $\wedge$
($ce.parentAS0$.**s**-<signal list item>**-seq**$.signalSet_0 \subseteq gc$.**s**-<signal list item>**-seq**$.signalSet_0$)

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.baseType*$_0$ **in**
(*td.channelDefinitionSet*$_0$ ≠ ∅) ⇒
(∀*gc*∈ <gate constraint>:∀*sig* ∈ *SIGNAL*$_0$:
(*gc.parentAS0*∈ *td.gateDefinitionSet*$_0$) ∧ (*sig*∈ *gc*.**s**-<signal list item>**-seq**.*signalSet*$_0$) ⇒
(∃*cp*∈ <channel path>:
(*cp.parentAS0*∈ *td.channelDefinitionSet*$_0$) ∧
((*gc.direction*$_0$=**in**)⇒
(*cp.origination*$_0$.**s**-<gate> = *gc.parentAS0*.**s**-<gate>) ∧
(*cp.origination*$_0$.**s-implicit**=**env**)) ∧
((*gc.direction*$_0$=**out**)⇒
(*cp.destination*$_0$.**s**-<gate> = *gc.parentAS0*.**s**-<gate>) ∧
(*cp.destination*$_0$.**s-implicit**=**env**)) ∧
(*sig*∈ *cp*.**s**-<signal list item>**-seq**.*signalSet*$_0$))) **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.direction*$_0$=**out**) ∧ (*gc'.direction*$_0$=**in**)∨(*gc.direction*$_0$=**in**)∧(*gc'.direction*$_0$=**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.surroundingScopeUnit*$_0$ **in**
∃*td'*∈ *TYPEDEFINITION*$_0$: ∃*gd'*∈ <textual gate definition>:
isSubtype$_0$(*td*, *td'*) ∧ (*gd'*∈ *td'.gateDefinitionSet*$_0$) ∧
(*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>:
isSubtype$_0$(*ec.surroundingScopeUnit*$_0$, *ec'.surroundingScopeUnit*$_0$) ∧
(*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.direction*$_0$=**out**) ∧ (*ec'.direction*$_0$=**in**))∨((*ec.direction*$_0$=**in**) ∧ (*ec'.direction*$_0$=**out**))) ∧
(*ec.parentAS0.parentAS0*.**s**-<gate>= *ec'.parentAS0.parentAS0*.**s**-<gate>) ⇒
(**let** *td* = *getEntityDefinition*$_0$(*ec*.**s**-<identifier>, *ec.surroundingScopeUnit*$_0$.*entityKind*$_0$) **in**
let *td'* = *getEntityDefinition*$_0$(*ec'*.**s**-<identifier>, *ec'.surroundingScopeUnit*$_0$.*entityKind*$_0$) **in**
(*td* = *td'*) ∨ *isSubtype*$_0$(*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 = *getEntityDefinition*$_0$(*gd*.**s**-<identifier>, **interface**)

The <interface identifier> of an <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>.

gateDefinitionSet$_0$(*td*: <agent type definition> ∪<composite state type definition>∪
<agent definition> ∪<composite state>): <gate in definition>-**set** =$_{def}$
td.localGateDefinitionSet$_0$∪*td.inheritedGateDefinitonSet*$_0$

localGateDefinitionSet$_0$(*td*: <agent type definition> ∪ <composite state type definition> ∪
<agent definition> ∪<composite state>): <gate in definition>-**set** =$_{def}$
{*gd*∈ <gate in definition> : *gd.surroundingScopeUnit*$_0$ = *td*}

inheritedGateDefinitonSet$_0$(*td*: <agent type definition> ∪ <composite state type definition>∪
<agent definition> ∪<composite state>): <gate in definition>-**set** =$_{def}$
let *sp*=*td.specialization*$_0$ **in**
if *sp*=*undefined* **then** ∅
else *sp*.**s**-<type expression>.*baseType*$_0$.*gateDefinitionSet*$_0$
endif
endlet

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

channelDefinitionSet$_0$(*td*: <agent type definition> ∪<agent definition>): <channel definition>-**set** =$_{def}$
td.localChannelDefinitionSet$_0$∪ *td.inheritedChannelDefinitionSet*$_0$

localChannelDefinitionSet$_0$(*td*: <agent type definition> ∪<agent definition>): <channel definition>-**set** =$_{def}$
{*cd*∈ <channel definition>: *cd.surroundingScopeUnit*$_0$ = *td*}

inheritedChannelDefinitionSet$_0$(*td*:<agent type definition>∪<agent definition>): <channel definition>-**set** =$_{def}$
let *sp*=*td.specialization*$_0$ **in**
if *sp*=*undefined* **then** ∅
else *sp*.**s**-<type expression>.*baseType*$_0$.*channelDefinitionSet*$_0$
endif
endlet

Get the identifiers of the kind *SIGNAL*$_0$.

signalSet$_0$(*sl*:<signal list item>*****): *SIGNAL*$_0$=$_{def}$
case *sl.head.idKind*$_0$ **of**
| {**signal**, **timer**, **remote procedure**, **remote variable**}=>{*sl.head*}∪ *sl.tail.signalSet*$_0$
| {**interface**} =>
let *fd* = *getEntityDefinition*$_0$(*sl.head*, **interface**) **in**
fd.usedSignalSet$_0$∪{*sd.identifier*$_0$: *sd*∈ *fd.definedSignalSet*$_0$}∪ *sl.tail.signalSet*$_0$
| {**signallist**} =>.**s**-<signal list item>-**seq**.*signalSet*$_0$∪
(**let** *sld* = *getEntityDefinition*$_0$(*sl.head*, **signallist**) **in**
signalSet$_0$(*sld*.**s**-<signal list item>-**seq**)∪*sl.tail.signalSet*$_0$ **endlet**)
| **otherwise** =>∅
endcase

2.5.7 Context parameters

Concrete Syntax

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

<formal context parameter> =
<agent type context parameter>
| <agent context parameter>
| <procedure context parameter>
| <remote procedure context parameter>
| <signal context parameter>
| <variable context parameter>
| <remote variable context parameter>
| <timer context parameter>
| <synonym context parameter>
| <sort context parameter>
| <exception context parameter>
| <composite state type context parameter>
| <gate context parameter>
| <interface context parameter>

Conditions on Concrete Syntax

∀*fcp*∈ *FORMALCONTEXTPARAMETER*$_0$:∀*acp*∈<actual context parameter>:
isContextParameterCorresponded$_0$(*fcp*, *acp*) ∧ (*acp*∈<primary>) ⇒
(*fcp*∈<synonym context parameter gen name>)

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

(∀*te*∈<type expression>: *te.baseType*$_0$∉ *FORMALCONTEXTPARAMETER*$_0$) ∧
(∀*fcp*∈ *FORMALCONTEXTPARAMETER*$_0$:
fcp.contextParameterAtleastDefinition$_0$∉ *FORMALCONTEXTPARAMETER*$_0$)

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 gen name> ∪<sort context parameter>∪
<composite state type context parameter> ∪<interface context parameter gen name> :
∀*acp*∈<actual context parameter>: *isContextParameterCorresponded*$_0$(*fcp*, *acp*) ⇒
(∀*td'*∈ *TYPEDEFINITION*$_0$: (*td'* = *fcp.contextParameterAtleastDefinition*$_0$) ⇒

$(td' \notin FORMALCONTEXTPARAMETER_0) \wedge \neg(isParameterizedType_0(td')) \wedge$
$(\exists td \in TYPEDEFINITION_0:$
$(td = getEntityDefinition_0(acp, td'.entityKind_0) \wedge$
$((td = td') \vee isSubtype_0(td, td'))))$

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
getUnboundFormalContextParameterList$_0$(*actParams*, *base.refersto*$_0$.**s-**<formal context parameter>)
≠ *empty*
=11=>
(**let** *nCPar* = *cPar* ∩
getUnboundFormalContextParameterList$_0$(*actParams*, *base.refersto*$_0$.**s-**<formal context parameter>)
in
let *nActPar* =
completeFormalContextParameter$_0$(*actParams*, *base.refersto*$_0$.**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
getUnboundFormalContextParameterList$_0$(*actParams*, *base.refersto*$_0$.**s-**<formal context parameter>)
≠ *empty*
=11=>
(**let** *nCPar* = *cPar* ∩
getUnboundFormalContextParameterList$_0$(*actParams*, *base.refersto*$_0$.**s-**<formal context parameter>)
in
let *nActPar* =
completeFormalContextParameter$_0$(*actParams*, *base.refersto*$_0$.**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
getUnboundFormalContextParameterList$_0$(*actParams*, *base.refersto*$_0$.**s-**<formal context parameter>)
≠ *empty*
=11=>
(**let** *nCPar* = *cPar* ∩
getUnboundFormalContextParameterList$_0$(*actParams*, *base.refersto*$_0$.**s-**<formal context parameter>)
in
let *nActPar* =
completeFormalContextParameter$_0$(*actParams*, *base.refersto*$_0$.**s-**<formal context parameter>)
in
<procedure heading>(*v*, *q*, *n*, *nCPar*, *vc*,
<specialization>(<type expression>(*b*, *nActPar*), *undefined*), *p*, *r*, *x*)
endlet)

<signal definition item>(*n*, *cPar*, *vc*, <specialization>(<type expression>(*base*, *actPar*), *), *p*)
provided
getUnboundFormalContextParameterList$_0$(*actParams*, *base.refersto*$_0$.**s-**<formal context parameter>)
≠ *empty*

=11=>
(**let** *nCPar* = *cPar* ⌢
getUnboundFormalContextParameterList$_0$(*actParams*, *base.refersto*$_0$.**s-**<formal context parameter>)
in
let *nActPar* =
completeFormalContextParameter$_0$(*actParams*, *base.refersto*$_0$.**s-**<formal context parameter>)
in
<signal definition item>(*n*,*nCPar*, *vc*, <specialization>(<type expression>(*b*, *nActPar*), *undefined**), *p*)
endlet)

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

Auxiliary Functions

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

contextParameterAtleastDefinition$_0$(*fcp*: *FORMALCONTEXTPARAMETER*$_0$): *ENTITYDEFINITION*$_0$ =$_{def}$
case *fcp* **of**
| <agent type context parameter> =>
if (*fcp*.**s-**<agent type constraint>∈ <identifier>) **then**
getEntityDefinition$_0$(*fcp*.**s-**<agent type constraint>.<identifier>, **agent type**)
else *undefined*
endif
| <agent context parameter> =>
if (*fcp*.**s-**<agent constraint>∈ <agent constraint gen atleast>) **then**
getEntityDefinition$_0$(*fcp*.**s-**<agent constraint>.**s-**<identifier>, **agent type**)
else *undefined*
endif
| <procedure context parameter> =>
if (*fcp*.**s-**<procedure constraint>∈ <identifier>) **then**
getEntityDefinition$_0$(*fcp*.**s-**<procedure constraint>, **procedure**)
else *undefined*
endif
| <composite state type context parameter> =>
if (*fcp*.**s-**<composite state type constraint>∈ <identifier>) **then**
getEntityDefinition$_0$(*fcp*.**s-**<composite state type constraint>,
state type)
else *undefined*
endif
| <signal context parameter gen name> =>
if (*fcp*.**s-**<signal constraint>∈ <identifier>) **then**
getEntityDefinition$_0$(*fcp*.**s-**<signal constraint>, **signal**)
else *undefined*
endif
| <sort context parameter> =>
if (*fcp*.**s-**<sort constraint>∈ <sort>) **then**
getEntityDefinition$_0$(*fcp*.**s-**<sort constraint>, **type**)
else *undefined*
endif
| <interface context parameter gen name> =>
if (*fcp*.**s-**<interface constraint> ≠ *undefined*) **then**
getEntityDefinition$_0$(*fcp*.**s-**<interface constraint>.**s-**<identifier>, **interface**)
else *undefined*
endif
otherwise *undefined*
endcase

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

$\forall$*fcp*$\in$ <agent type context parameter>:$\forall$*acp*$\in$ <actual context parameter>:
isContextParameterCorresponded$_0$(*fcp*, *acp*) $\wedge$ (*fcp*.**s-**<agent type constraint> $\neq$ *undefined*) $\Rightarrow$
(**let** *td*=*getEntityDefinition*$_0$(*acp*, **agent type**) **in**
($\exists$*td'*$\in$ <agent type definition> :
(*td'*=*fcp.contextParameterAtleastDefinition*$_0$) $\wedge$
(*td.agentLocalFormalParameterList*$_0$=*empty*) $\wedge$ *isSubtype*$_0$(*td*,*td'*)) $\vee$
((*td.entityKind*$_0$ = *fcp.entityKind*$_0$) $\wedge$
(**let** *pl* = *td.agentFormalParameterList*$_0$ **in**
let *sl* = *fcp*.**s-**<agent type constraint>.*agentSignatureSortList*$_0$ **in**
(*pl.length* = *sl.length*) $\wedge$
($\forall$*i*$\in$ 1..*pl.length*: *isSameSort*$_0$(*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>.

agentSignatureSortList$_0$(*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>.

agentFormalParameterList$_0$(*td*: <agent type definition> $\cup$<agent definition> $\cup$
<composite state type definition>$\cup$<composite state>): <name>* =$_{def}$
td.agentLocalFormalParameterList$_0$ $\frown$ *td.agentInheritedFormalParameterList*$_0$

agentLocalFormalParameterList$_0$(*td*: <agent type definition> $\cup$<agent definition> $\cup$
<composite state type definition>$\cup$<composite state>): <name>* =$_{def}$
let *fp*=*take*({*fp*$\in$ <agent additional heading>$\cup$<composite state type heading>$\cup$<composite state heading>:
fp.surroundingScopeUnit$_0$ = *td*}) **in**
< *psl*.**s-**<name> | *psl* **in** *fp*.**s-**<formal agent parameter>**-seq** >
endlet

agentInheritedFormalParameterList$_0$(*td*: <agent type definition> $\cup$<agent definition> $\cup$
<composite state type definition>$\cup$<composite state>): <name>* =$_{def}$
let *sp*=*td.specialization*$_0$ **in**
if (*sp* = *undefined*) **then** *empty*
else *sp*.**s-**<type expression>.*baseType*$_0$.*agentFormalParameterList*$_0$
endif
endlet

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

isContextParameterCorresponded$_0$(*fcp*: *FORMALCONTEXTPARAMETER*$_0$, *acp*: <actual context parameter>):
BOOLEAN=$_{def}$
let *fcpl* = *fcp.surroundingScopeUnit*$_0$.*formalContextParameterList*$_0$**in**
let *acpl* = *parentAS0ofKind*(*acp*, <type expression>).*actualContextParameterList*$_0$ **in**
(*fcpl.length*= *acpl.length*)∧
(∃!*i*∈ 1..*fcpl.length*: (*fcpl*[*i*]=*fcp*)∧(*acpl*[*i*]=*acp*)
endlet

2.5.7.2 Agent Context Parameter

Concrete Syntax

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

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

<agent constraint gen atleast> :: [**atleast**] <agent type<identifier>

<agent signature> :: <sort>+

Conditions on Concrete Syntax

∀*fcp*∈ <agent context parameter> :∀*acp*∈ <actual context parameter>:
isContextParameterCorresponded$_0$(*fcp*, *acp*) ⇒
(**let** *td* = *getEntityDefinition*$_0$(*acp*, **agent**) **in**
let *td′* = *fcp.contextParameterAtleastDefinition*$_0$ **in**
((*fcp*.**s**-<agent constraint>.**s-atleast** ≠*undefined*) ⇒
isSubtype$_0$(*td*, *td′*) ∧ (*td.agentLocalFormalParameterList*$_0$ = *empty*)) ∧
((*fcp*.**s**-<agent constraint> **s-atleast** =*undefined*) ⇒ (*td* = *td′*)) ∧
((*fcp*.**s**-<agent constraint> ∈ <agent signature>) ⇒
(*getEntityDefinition*$_0$(*acp*, **agent**).*entityKind*$_0$ = *fcp.entityKind*$_0$) ∧
(**let** *pl* = *td.agentFormalParameterList*$_0$ **in**
let *sl* = *fcp*.**s**-<agent constraint>.*agentSignatureSortList*$_0$ **in**
(*pl.length* = *sl.length*) ∧
(∀*i*∈ 1..*pl.length*: *isSameSort*$_0$(*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 <agent signature>, and both definitions have the same *Agent-kind*.

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>:
isContextParameterCorresponded$_0$(*fcp*, *acp*) ⇒
(**let** *td* = *getEntityDefinition*$_0$(*acp*, **procedure**) **in**
((*fcp*.**s**-<procedure constraint>∈ <identifier>) ⇒
(**let** *td′* = *fcp.contextParameterAtleastDefinition*$_0$ **in**
isDirectSubType$_0$(*td*, *td′*) **endlet**)) ∧

((*fcp*.**s**-<procedure constraint>∈ <procedure signature>) ⇒
(**let** *fpl* = *td.procedureFormalParameterList*$_0$ **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**}) ⇒
isSameSort$_0$(*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*)∧*isSameResult*$_0$(*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.

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>:
isContextParameterCorresponded$_0$(*fcp*,*acp*) ⇒
(**let** *ps* = *getEntityDefinition*$_0$(*acp*, **remote procedure**).**s**-<procedure signature> **in**
let *ps'*= *fcp*.**s**-<procedure signature> **in**
isSameProcedureSignature$_0$(*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.

isSameProcedureSignature$_0$(*ps*, *ps'*: <procedure signature>): *BOOLEAN* =$_{\text{def}}$
let *fpl* = *ps.procedureSignatureParameterList*$_0$ **in**
let *fpl'* = *ps'.procedureSignatureParameterList*$_0$ **in**
(*fpl*. *length* = *fpl'.length*) ∧
(∀*i*∈ 1..*fpl.length*:
isSameSort$_0$(*fpl*[*i*].**s**-<sort>, *fpl'*[*i*].**s**-<sort>) ∧
(*fpl*[*i*].**s**-<parameter kind> = *fpl'*[*i*].**s**-<parameter kind>))∧
isSameResult$_0$(*ps*.**s**-<result>, *ps'*.**s**-<result>) ∧
isSameRaises$_0$(*ps*.**s**-<raises>, *ps'*.**s**-<raises>)
endlet

2.5.7.5 Signal Context Parameter

Concrete Syntax

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

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

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

<signal signature> = <sort>+

Conditions on Concrete Syntax

∀*fcp*∈ <signal context parameter gen name> :∀*acp*∈ <actual context parameter>:
isContextParameterCorresponded$_0$(*fcp*, *acp*) ⇒
(**let** *sd* = *getEntityDefinition*$_0$(*acp*, **signal**) **in**
((*fcp*.**s**-<signal constraint>∈ <identifier>) ⇒
(**let** *sd′* =*fcp.contextParameterAtleastDefinition*$_0$ **in**
isSubtype$_0$(*sd*, *sd′*) **endlet**)) ∧
((*fcp*.**s**-<signal constraint>∈ <signal signature>) ⇒
isSameSortList$_0$(*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.

2.5.7.6 Variable Context Parameter

Concrete Syntax

<variable context parameter> :: <variable context parameter gen name>+

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

<variable constraint> = <sort>

Conditions on Concrete Syntax

∀*fcp*∈ <name>:∀*acp*∈ <actual context parameter>:
(*fcp.parentAS0*∈ <variable context parameter gen name>) ∧
(*fcp.parentAS0.parentAS0*∈ <variable context parameter>) ∧
isContextParameterCorresponded$_0$(*fcp*, *acp*) ⇒
(**let** *vd* = *getEntityDefinition*$_0$(*acp*, **variable**) **in**
isSameSort$_0$(*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.

2.5.7.7 Remote Variable Context Parameter

Concrete Syntax

<remote variable context parameter>::
<remote variable context parameter gen name>+

<remote variable context parameter gen name>::
<remote variable<name>+ <variable constraint>

Conditions on Concrete Syntax

∀*fcp*∈ <name>:∀*acp*∈ <actual context parameter>:
(*fcp.parentAS0*∈ <remote variable context parameter gen name>) ∧
(*fcp.parentAS0.parentAS0*∈ <remote variable context parameter>) ∧
isContextParameterCorresponded$_0$(*fcp*, *acp*) ⇒

(**let** *vd* = *getEntityDefinition*$_0$(*acp*, **remote variable**) **in**
isSameSort$_0$(*fcp.parentAS0*.**s**-<sort>, *vd*.**s**-<sort>)
endlet)

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

2.5.7.8 Timer Context Parameter

Concrete Syntax

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

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

<timer constraint> = <sort>+

Conditions on Concrete Syntax

∀*fcp*∈ <timer context parameter gen name> :∀*acp*∈ <actual context parameter>:
isContextParameterCorresponded$_0$(*fcp*, *acp*) ⇒
(**let** *td* = *getEntityDefinition*$_0$(*acp*, **timer**) **in**
(*fcp*.**s**-<timer constraint> ≠*undefined*) ⇒
isSameSortList$_0$(*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.

2.5.7.9 Synonym Context Parameter

Concrete Syntax

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

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

<synonym constraint> = <sort>

Conditions on Concrete Syntax

∀*fcp*∈ <synonym context parameter gen name> :∀*acp*∈ <actual context parameter>:
isContextParameterCorresponded$_0$(*fcp*, *acp*) ⇒
(**let** *sd* = *getEntityDefinition*$_0$(*acp*, **synonym**) **in**
isSameSort$_0$(*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.

2.5.7.10 Sort Context Parameter

Concrete Syntax

<sort context parameter> :: [<data type kind>] <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>:
isContextParameterCorresponded$_0$(*fcp*, *acp*) ∧ (*fcp*.**s**-<sort constraint>≠*undefined*)⇒
(**let** *td* = *getEntityDefinition*$_0$(*acp*, **type**).*derivedDateType*$_0$ **in**
(*fcp*.**s**-<sort constraint>∈ <sort>)⇒
(**let** *td′* = *fcp.contextParameterAtleastDefinition*$_0$ **in**
isSubtype$_0$(*td*, *td′*)∧
(∀*td1*∈ <data type definition>∪{*td*}:*isSubtype*$_0$(*td1*, *td′*)⇒
td1.**s**-<data type specialization>.**s**-<renaming>= *undefined*)
endlet)) ∧
((*fcp*.**s**-<sort constraint>∈ <sort signature>)⇒
(∀*ls*∈ <literal signature>: (*ls.parentAS0*= *fcp*.**s**-<sort constraint>)⇒
∃*ls′*∈ <literal signature>:
(*ls′.surroundingScopeUnit*$_0$= *td*) ∧ *isSameLiteralSignature*$_0$(*ls*,*ls′*)) ∧
(∀*os*∈ <operation signature> :
(*os.parentAS0*= *fcp*.**s**-<sort constraint>)⇒
∃*os′*∈ <operation signature> :
(*os′.surroundingScopeUnit*$_0$= *td*) ∧(*os.virtuality*$_0$ = *os′. virtuality*$_0$) ∧
(*os.visibility*$_0$ = *os′.visibility*$_0$) ∧ *isSameOperationSignature*$_0$(*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.

derivedDateType$_0$(*sd*: <syntype definition>∪<data type definition>):<data type definition> =$_{def}$
if (*sd*∈ <syntype definition>) **then** *sd.parentDataType*$_0$
else *sd*
endif

Get the parent data type definition of a syntype definition.

parentDataType$_0$(*sd*: <syntype definition>):<data type definition> =$_{def}$
if (*sd*.**s**-<parent sort identifier> = *undefined*) **then** *sd*
else
let *pd* = *getEntityDefinition*$_0$(*sd*.**s**-<parent sort identifier>, **type**) **in**
if (*pd*∈ <data type definition>) **then** *pd*
else *pd.parentDataType*$_0$
endif
endlet
endif

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

isSameLiteralSignature$_0$(*ls*: <literal signature>, *ls′*: <literal signature>): *BOOLEAN*=$_{def}$
((*ls*∈ <literal name>) ∧ (*ls′*∈ <literal name>) ⇒ (*ls* = *ls′*)) ∧
((*ls*∈ <name class literal>) ∧ (*ls′*∈ <name class literal>) ⇒ *isSameNameClassLiteral*$_0$(*ls*, *ls′*)) ∧
((*ls*∈ <named number>) ∧ (*ls′*∈ <named number>) ⇒
(*ls*.**s**-<literal name> = *ls′*.**s**-<literal name>) ∧
(*ls*.**s**-<simple expression>.*value*$_0$= *ls′*.**s**-<simple expression>.*value*$_0$))

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

isSameOperationSignature$_0$(*os*: <operation signature> , *os′*: <operation signature>): *BOOLEAN*=$_{\text{def}}$
(*os*.**s-implicit**∈ <operation name>⇒*os*.**s-implicit** = *os′*.**s- implicit**) ∧
(*os*.**s-implicit**∈ <name class operation>⇒*isSameNameClassOperation*$_0$(*os*.**s- implicit** , *os′*.**s- implicit**)) ∧
(**let** *fpl* = *os.operationSignatureParameterList*$_0$ **in**
let *fpl′* = *os′.operationSignatureParameterList*$_0$ **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>) ∧
isSameSort$_0$(*fpl*[*i*].**s-**<formal parameter>.**s-**<sort>, *fpl′*[*i*].**s-**<formal parameter>.**s-**<sort>)) ∧
isSameResult$_0$(*os*.**s-**<result>, *os′*.**s-**<result>) ∧
isSameRaises$_0$(*os*.**s-**<raises>, *os′*.**s-**<raises>)
endlet)

2.5.7.11 Exception Context Parameter

Concrete Syntax

<exception context parameter>::=
<exception context parameter gen name>+

<exception context parameter gen name> ::
<exception><name> [<exception constraint>]

<exception constraint> =<sort>+

Conditions on Concrete Syntax

∀*fcp*∈ <exception context parameter gen name> :∀*acp*∈ <actual context parameter>:
isContextParameterCorresponded$_0$(*fcp*, *acp*) ∧ (*fcp*.**s-**<exception constraint> ≠ *undefined*) ⇒
(**let** *ed* = *getEntityDefinition*$_0$(*acp*, **exception**) **in**
isSameSortList$_0$(*fcp*.**s-**<exception constraint>.**s-**<sort>**-seq** , *ed*.**s-**<sort>**-seq**)
endlet)

An actual exception parameter must identify an exception with the same signature.

2.5.7.12 Composite State Context Parameter

Concrete Syntax

<composite state 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>+

Conditions on Concrete Syntax

∀*fcp*∈ <composite state type context parameter> :∀*acp*∈ <actual context parameter>:
isContextParameterCorresponded$_0$(*fcp*, *acp*)∧(*fcp*.**s-**<composite state type constraint>≠ *undefined*) ⇒
(**let** *td* = *getEntityDefinition*$_0$(*acp*, **state type**) **in**
((*fcp*.**s-**<composite state type constraint>∈ <identifier>) ⇒
(**let** *td′* = *fcp.contextParameterAtleastDefinition*$_0$ **in**
(*td.agentLocalFormalParameterList*$_0$ = *empty*) ∧ *isSubtype*$_0$(*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.agentFormalParameterList*$_0$ **in**
(*sl.length* = *pl.length*) ∧
(∀*i*∈ 1..*sl.length*: *isSameSort*$_0$(*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>.

2.5.7.13 Gate Context Parameter

Concrete Syntax

<gate context parameter> :: <gate> <gate constraint> [<gate constraint>]

Conditions on Concrete Syntax

∀*fcp*∈ <gate context parameter>:∀*acp*∈ <actual context parameter>:
isContextParameterCorresponded$_0$(*fcp*, *acp*) ⇒
(**let** *gd*=*getEntityDefinition*$_0$(*acp*, **gate**) **in**
(*gd*.**s**-<gate> = *fcp*.**s**-<gate>) ∧
(∀*gc*∈ <gate constraint>:∀*gc'*∈ <gate constraint>:
(*gc.parentAS0* = *gd*) ∧ (*gc'.parentAS0*= *fcp*) ∧
((*gc.direction*$_0$=**out**) ∧ (*gc'. direction*$_0$=**out**) ⇒
gc'.**s**-<sort>**-seq** *.signalSet*$_0$) ⊆*gc*.**s**-<sort>**-seq** *.signalSet*$_0$) ∧
((*gc*. *direction*$_0$=**in**) ∧ (*gc'*. *direction*$_0$=**in**) ⇒
gc.**s**-<sort>**-seq** *.signalSet*$_0$⊆*gc'*.**s**-<sort>**-seq** *.signalSet*$_0$)
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.

2.5.7.14 Interface Context Parameter

Concrete Syntax

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

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

<interface constraint> :: <interface<identifier>

Conditions on Concrete Syntax

∀*fcp*∈ <interface context parameter gen name> :∀*acp*∈ <actual context parameter>:
isContextParameterCorresponded$_0$(*fcp*, *acp*) ∧ (*fcp*.**s**-<interface constraint>≠ *undefined*) ⇒
(∃*td*∈ <interface definition>:
(*td* =*getEntityDefinition*$_0$(*acp*.**s**-<identifier>, **interface**)) ∧
isSubtype$_0$(*td*, *fcp.contextParameterAtleastDefinition*$_0$))

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>).

2.5.8 Specialization

2.5.8.1 Adding properties

Concrete Syntax

<specialization> :: <type expression>

Mapping to Abstract Syntax

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

Auxiliary Functions

The function $specialization_0$ is used to get the <specialization> part of an entity definition.

$specialization_0$(*def*: $ENTITYDEFINITION_0$): <specialization> $=_{def}$
take({ *s*∈ <specialization>∪<data type specialization>∪<interface specialization>:
$s.surroundingScopeUnit_0 = def$})

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 definitions>∪
<composite state type definition>:
$isVirtualType_0(td) \Rightarrow td.virtualTypeAtleastDefinition_0.entityKind_0 = td.entityKind_0$

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 definitions>∪
<composite state type definition>:
$td.isVirtualType_0 \Rightarrow$
$\neg(td.isParameterizedType_0) \wedge \neg(isParameterizedType_0(td.virtualTypeAtleastDefinition_0))$

A virtual type and its constraints cannot have context parameters.

∀*vc*∈ <virtuality constraint>: $isVirtualType_0(vc.surroundingScopeUnit_0)$

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 definitions>:
$(r.referencedDefinition_0 = d) \wedge (r.virtuality_0 \neq undefined) \wedge (d.virtuality_0 \neq undefined) \Rightarrow$
$(r.virtuality_0 = d.virtuality_0)$

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>: $isVirtualType_0(td) \Rightarrow$
(**let** $td' = td.virtualTypeAtleastDefinition_0$ **in**
(**let** $fpl = td.agentFormalParameterList_0$ **in**
let $fpl' = td'.agentFormalParameterList_0$ **in**
$isSameAgentFormalParameterList_0(fpl, fpl')$ **endlet**) ∧

(∀*gd'*∈ <gate in definition> : *gd'*∈ *td'.gateDefinitionSet*$_0$ ⇒
∃*gd*∈ <gate in definition> : (*gd*∈ *td.gateDefinitionSet*$_0$) ∧ *isSameGate*$_0$(*gd*, *gd'*)) ∧
(∀*fd'*∈ <interface definition>: *fd'*∈ *td'.interfaceDefinitionSet*$_0$ ⇒
∃*fd*∈ <interface definition>: (*fd*∈ *td.interfaceDefinitionSet*$_0$) ∧ *isSameInterface*$_0$(*fd*, *fd'*)) ∧
(∀*ed*∈ *ENTITYDEFINITION*$_0$: *ed.surroundingScopeUnit0* = *td*⇒
∃*ed'*∈ *ENTITYDEFINITION*$_0$: (*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>: *isVirtualType*$_0$(*td*) ⇒
(**let** *td'* = *td.virtualTypeAtleastDefinition*$_0$ **in**
∀*scp'*∈ <state connection points>: *scp'*∈ *td'.stateConnectionPointSet*$_0$ ⇒
∃*scp*∈ <state connection points>:
(*scp*∈ *td.stateConnectionPointSet*$_0$) ∧ *isSameStateConnectionPoint*$_0$(*scp*, *scp'*) **endlet**)

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

∀*td*∈ <procedure definitions>: *isVirtualType*$_0$(*td*) ⇒
(**let** *fpl* = *td.procedureFormalParameterList*$_0$ **in**
let *fpl'* = *td.virtualTypeAtleastDefinition*$_0$.*procedureFormalParameterList*$_0$ **in**
isSameProcedureFormalParameterList$_0$(*fpl*, *fpl'*) **endlet**)

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

∀*td*∈ <block type definition>∪<process type definition>∪<procedure definitions>∪
<composite state type definition>:
isVirtualType$_0$(*td*) ⇒
(∀*sp*∈ <specialization>:∀*vc*∈ <virtuality constraint>:
(*sp.surroundingScopeUnit*$_0$ = *td*) ∧ (*vc.surroundingScopeUnit*$_0$ = *td*) ⇒
((*td.virtualTypeInheritsDefinition*$_0$= *td.virtualTypeAtleastDefinition*$_0$) ∨
isSubtype$_0$(*td.virtualTypeInheritsDefinition*$_0$, *td.virtualTypeAtleastDefinition*$_0$)))

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 definitions>∪
<composite state type definition>:
isRedefinedType$_0$(*td*) ∧ (*td'*=*td.superCounterpart*$_0$) ⇒
(**let** *sp*=*td.specialization*$_0$**in**
let *vc* = *take*({*vc*∈ <virtuality constraint>: *vc.surroundingScopeUnit*$_0$ = *td'*}) **in**
(*sp*≠*undefined*) ∧ (*vc*=*undefined*) ⇒
isSubtype$_0$(*td.virtualTypeInheritsDefinition*$_0$, *td'.virtualTypeAtleastDefinition*$_0$) **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>, a <procedure definitions> or a <composite state type definition> is a virtual type.

isVirtualType$_0$(*td*: <block type definition>∪<process type definition>∪<procedure definition>∪
<composite state type definition>): *BOOLEAN*=$_{\text{def}}$
td.virtuality$_0$∈ {**virtual**, **redefined**}

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

isRedefinedType$_0$(*td*: <block type definition>∪<process type definition>∪<procedure definitions>
∪<composite state type definition>): *BOOLEAN*=$_{\text{def}}$
td.*virtuality*$_0$∈ {**redefined**, **finalized**}

Get the virtuality for a definition.

virtuality$_0$(*td*: *DefinitionAS0*): {**virtual**, **redefined**, **finalized**} =$_{\text{def}}$
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>
| <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>
| <interface reference> => *d*.**s**-<virtuality>
| <operation signature> => *d*.**s**-<operation preamble>.**s**-<virtuality>
| <start>∪<input part>∪<priority input>∪<save part>∪<spontaneous transition>∪
<continuous signal>∪<connect part>∪<handle>∪<default initialization> => *d*.**s**-<virtuality>
| <statement list> => *d*.*parentAS0*.**s**-<virtuality>
otherwise *undefined*
endcase

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

virtualTypeAtleastDefinition$_0$(*td*: <block type definition>∪<process type definition>∪
<procedure definitions>∪<composite state type definition>): <block type definition>∪
<process type definition>∪<procedure definitions>∪<composite state type definition> =$_{\text{def}}$
let *vc*=*take*({*vc*∈ <virtuality constraint>: *vc*.*surroundingScopeUnit*$_0$ = *td*}) **in**
if *vc* ≠ *undefined* **then** *getEntityDefinition*$_0$(*vc*, *td*.*entityKind*$_0$) **else** *td* **endif**
endlet

Determine if two agent formal parameter lists are the same.

isSameAgentFormalParameterList$_0$(*fpl*: <name>*, *fpl'*:<name>*): *BOOLEAN*=$_{\text{def}}$
(*fpl*.*length* = *fpl'*.*length*) ∧
(∀*i*∈ 1..*fpl*.*length*: (*fpl*[*i*] = *fpl'*[*i*]) ∧ *isSameSort*$_0$(*fpl*[*i*].*parentAS0*.**s**-<sort>, *fpl'*[*i*].*parentAS0*$_0$.**s**-<sort>))

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

isSameGate$_0$(*gd*: <gate in definition> , *gd'*: <gate in definition>): *BOOLEAN*=$_{\text{def}}$
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*.*direction*$_0$ = *gc'*.*direction*$_0$)∧
(*gc*.**s**-<textual endpoint constraint> =*gc'*.**s**-<textual endpoint constraint>)∧
isSameSortList$_0$(*gc*.**s**-<sort>**-seq** , *gc'*.**s**-<sort>**-seq**))∧
(∀*gc'*∈ *gd'*.**s**-<gate constraint>:∃*gc*∈ *gd*.**s**-<gate constraint>:
(*gc*.*direction*$_0$ = *gc'*.*direction*$_0$)∧
(*gc*.**s**-<textual endpoint constraint> =*gc'*.**s**-<textual endpoint constraint>)∧
isSameSortList$_0$(*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.

isSameInterface$_0$(*id*:<interface definition>, *id′*:<interface definition>): *BOOLEAN*=$_{\text{def}}$
(*id.virtuality*$_0$ = *id′.virtuality*$_0$)∧
(*id.entityName*$_0$ = *id′.entityName*$_0$)∧
(*id.entityDefinitionSet*$_0$= *id′.entityDefinitionSet*$_0$)

Get all the entity definitions defined in a scope unit.

entityDefinitionSet$_0$(*su*: *SCOPEUNIT*$_0$): *ENTITYDEFINITION*$_0$ =$_{\text{def}}$
{*ed*∈ *ENTITYDEFINITION*$_0$: *isDefinedIn*$_0$(*ed*, *su*)}

Get all the interface definitions defined in a scope unit.

interfaceDefinitionSet$_0$(*d*: *SCOPEUNIT*$_0$):<interface definition>-**set** =$_{\text{def}}$
{*fd*∈ <interface definition>: *isDefinedIn*$_0$(*fd*, *d*)}

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

stateConnectionPointSet$_0$(*td*:<composite state type definition>∪<composite state>):
<state connection points>-**set** =$_{\text{def}}$
{*scp*∈ <state connection points>:
(*scp.parentAS0*=*td*)∧
(∃*td′*∈ <composite state type definition>: *isSubtype*$_0$(*td*, *td′*)∧ (*scp.parentAS0*=*td′*))}

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

isSameStateConnectionPoint$_0$(*scp*: <state connection points>, *scp′*: <state connection points>): *BOOLEAN*=$_{\text{def}}$
{*n*∈ <name>: *isAncestorAS0*(*scp*,*n*)} = {*n′*∈ <name>: *isAncestorAS0*(*scp′*,*n*)}

Determine if two procedure formal parameter lists are the same.

isSameProcedureFormalParameterList$_0$(*fpl*: <name>*, *fpl′*: <name>*): *BOOLEAN*=$_{\text{def}}$
(*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*]) ∧ *isSameSort*$_0$(*fpl*[*i*].*parentAS0*.**s**-<sort>, *fpl′*[*i*].*parentAS0*.**s**-<sort>))

Get the entity definition specialised by a virtual type.

virtualTypeInheritsDefinition$_0$(*td*: <block type definition>∪<process type definition>∪
<procedure definitions>∪<composite state type definition>): <block type definition>∪
<process type definition>∪<procedure definitions>∪<composite state type definition>=$_{\text{def}}$
let *sp*=*td.specialization*$_0$ **in**
if (*sp* ≠ *undefined*) **then** *sp*.**s**-<type expression>.*baseType*$_0$
else
let *vc*=*take*({*vc*∈ <virtuality constraint>: *vc.surroundingScopeUnit*$_0$ = *td*}) **in**
case *td.virtuality*$_0$ **of**
| **virtual** => **if** (*vc* = *undefined*) **then** *undefined* **else** *td.virtualTypeAtleastDefinition*$_0$ **endif**
| **redefined** =>
if (*vc* = *undefined*) **then** *td.superCounterpart*$_0$.*virtualTypeAtleastDefinition*$_0$
else *td.virtualTypeAtleastDefinition*$_0$
endif
| **finalized** => *td.superCounterpart*$_0$.*virtualTypeAtleastDefinition*$_0$
endcase
endlet
endif
endlet

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

superCounterpart$_0$(*td*: <block type definition>∪<process type definition>∪<procedure definitions>∪
<composite state type definition>∪<operation definition>∪<operation signature>):
(<block type definition>∪<process type definition>∪<procedure definitions>∪
<composite state type definition>∪<operation definition>)**-set** =$_{def}$
{*td'*∈ <block type definition>∪<process type definition>∪<procedure definitions>∪
<composite state type definition>∪<operation definition>∪<operation signature> :
isSuperCounterpart$_0$(*td'*, *td*)}

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

isSuperCounterpart$_0$(*td*: *DefinitionAS0*, *td'*: <block type definition>∪<process type definition>∪
<procedure definitions>∪<composite state type definition>∪<operation definition>): *BOOLEAN* =$_{def}$
(*td*.*name*$_0$= *td'*.*name*$_0$) ∧ (*td*.*kind*$_0$= *td'*.*kind*$_0$) ∧
(*td*.*virtuality*$_0$∈ {**virtual**, **redefined**})∧(*td'*.*virtuality*$_0$∈ {**redefined**, **finalized**}) ∧
isDirectSubType$_0$(*td'*.*surroundingScopeUnit*$_0$, *td*.*surroundingScopeUnit*$_0$)

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*.*startSet*$_0$⇒*s*.*virtuality*$_0$ = *undefined*) ∧
(∀*s*∈ <state>: (*s*∈ *ad*.*stateSet*$_0$)⇒
(*s*.**s-**<input part>.*virtuality*$_0$ = *undefined*) ∧
(*s*.**s-**<priority input>.*virtuality*$_0$ = *undefined*) ∧
(*s*.**s-**<save part>.*virtuality*$_0$ = *undefined*) ∧
(*s*.**s-**<spontaneous transition>.*virtuality*$_0$ = *undefined*) ∧
(*s*.**s-**<continuous signal>.*virtuality*$_0$ = *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*.*virtuality*$_0$≠*undefined*)}| ≤ 1

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

(∀*ip*∈ <input part>:
(*ip*.*virtuality*$_0$≠ *undefined*) ⇒ *ip*.**s-**<input list>∈ <asterisk input list>) ∧
(∀*sp*∈ <save part>:
(*sp*.*virtuality*$_0$ ≠ *undefined*) ⇒ *sp*.**s-**<save list>∈ <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.

startSet$_0$(*td*: <agent definition> ∪<textual typebased agent definition> ∪<agent type definition> ∪
<composite state> ∪\<typebased composite state>∪<composite state type definition>∪
<textual typebased state partition definition> ∪<state partition>∪
<procedure definitions>):<start>**-set** =$_{def}$
case *td* **of**
| <agent definition>∪<agent type definition>∪<composite state type definition>∪<procedure definitions>
=> *td*.*localStartSet*$_0$∪*td*.*inheritedStartSet*$_0$
| <textual typebased agent definition> ∪<textual typebased state partition definition> =>
let *te*=*take*({*te*∈ <type expression>: *te*.*parentAS0*.*parentAS0* = *td*}) **in**
te.*baseType*$_0$.*startSet*$_0$
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.startSet*$_0$: *sp*∈ <state partition>∧*sp.parentAS0*= *td*.**s-**<composite state structure>.**s-implicit** }
endif
| <typebased composite state> => *td*.**s-**<type expression>.*baseType*$_0$.*startSet*$_0$
| <composite state reference> => *td.referencedDefinition*$_0$.*startSet*$_0$
otherwise ∅
endcase

localStartSet$_0$(*td*: <agent definition> ∪<agent type definition> ∪<composite state type definition>∪
<procedure definitions>):<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-**<agent body gen start>.**s-**<start>}
else *td*.**s-**<agent structure>.**s-implicit**.**s-**<state partition>.*startSet*$_0$
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.startSet*$_0$: *sp*∈ <state partition>∧*sp.parentAS0=td*.**s-**<composite state structure>.**s-implicit** }
endif
| <procedure definitions> =>
if *td*.**s-implicit**∈ <procedure body> **then** { *td*.**s-implicit**.**s-**<start>}
else ∅
endif
otherwise ∅
endcase

inheritedStartSet$_0$(*td*: <agent definition> ∪<agent type definition> ∪<composite state type definition> ∪
<procedure definitions>): <start>**-set** =$_{def}$
let *sp=td.specialization*$_0$ **in**
if *sp=undefined* **then** ∅
else *sp*.**s-**<type expression>.*baseType*$_0$.*startSet*$_0$
endif
endlet

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

stateSet$_0$(*td*: <agent definition> ∪<textual typebased agent definition> ∪<agent type definition> ∪
<composite state> ∪<typebased composite state>∪<composite state type definition>∪
<procedure definitions>):<state>**-set** =$_{def}$
case *td* **of**
| <agent definition> ∪<agent type definition> ∪<composite state type definition>∪<procedure definitions>
=> *td.localStateSet*$_0$∪*td.inheritedStateSet*$_0$
| <textual typebased agent definition> ∪<textual typebased state partition definition> =>
let *te* = *take*({*te* ∈ <type expression>: *te.parentAS0.parentAS0 = td*}) **in**
te.baseType$_0$.*stateSet*$_0$
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.stateSet*$_0$: *sp*∈ <state partition>∧*sp.parentAS0=td*.**s-**<composite state structure>.**s-implicit** }
endif
| <typebased composite state> => *td*.**s-**<type expression>.*baseType*$_0$.*stateSet*$_0$
| <composite state reference> => *td.referencedDefinition*$_0$.*stateSet*$_0$
otherwise ∅
endcase

localStateSet$_0$(*td*: <agent definition> ∪<agent type definition> ∪<composite state type definition>∪
<procedure definitions>): <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 partition>.**s-implicit**.$stateSet_0$
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*. $stateSet_0$: *sp*∈ <state partition>∧*sp.parentAS0*=*td*.**s-**<composite state structure>.**s-implicit** }
endif
| <procedure definitions> =>
if *td*.**s-implicit**∈ <procedure body> **then** { *s*∈ <state>: *s.parentAS0*= *td*.**s-implicit**}
else ∅
endif
otherwise ∅
endcase

$inheritedStateSet_0$(*td*: <agent definition> ∪<agent type definition> ∪<composite state type definition>
∪<procedure definitions>):<state>**-set** =$_{def}$
let *sp*=*td*.$specialization_0$ **in**
if *sp*=*undefined* **then** ∅
else *sp*.**s-**<type expression>.$baseType_0$.$stateSet_0$
endif
endlet

2.5.8.4 Virtual Method

Conditions on Concrete Syntax

∀*os*∈ <operation signature> :
(*os*.$entityKind_0$=**method**) ∧ (*os*.$virtuality_0$∈ {**redefined**, **finalized**}) ⇒
∃*os′*∈ <operation signature> : (*os′* ∈ *os*.$superCounterpart_0$) ∧
$isOperationSignatureCompatible_0$(*os*, *os′*)∧
(**let** *fpl* = *os*.$operationSignatureParameterList_0$ **in**
let *fpl′* = *os′*.$operationSignatureParameterList_0$ **in**
∀*i*∈ 1..*fpl*.*length*:
(*fpl*[*i*].**s-**<formal parameter>.**s-**<parameter kind>=
fpl′[*i*].**s-**<formal parameter>.**s-**<parameter kind>)∧
((*fpl*[*i*].**s-**<argument virtuality>≠*undefined*)⇒(*fpl′*[*i*].**s-**<argument virtuality>≠*undefined*))
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>. A redefinition of a virtual method must not add <argument virtuality> to any <argument> of the inherited <operation signature>.

Auxiliary Functions

Determine if two <operation signature>s are compatible.

$isOperationSignatureCompatible_0$(*os*: <operation signature> , *os′*: <operation signature>): *BOOLEAN*=$_{def}$
$isSortCompatible_0$(*os*.**s-**<result>.**s-**<sort>, *os′*.**s-**<result>.**s-**<sort>) ∧
(**let** *fpl* = *os*.$operationSignatureParameterList_0$ **in**
let *fpl′* = *os′*.$operationSignatureParameterList_0$ **in**
(*fpl*.*length* = *fpl′*.*length*) ∧
(∀*i*∈ 1..*fpl*.*length*:
((*fpl*[*i*].**s-**<argument virtuality>≠*undefined*)⇒
$isSortCompatible_0$(*fpl*[*i*].**s-**<formal parameter>.**s-**<sort>,
fpl′[*i*].**s-**<formal parameter>.**s-**<sort>)) ∧

((*fpl*[*i*].**s-**<argument virtuality>=*undefined*)⇒
isSameSort$_0$(*fpl*[*i*].**s-**<formal parameter>.**s-**<sort>, *fpl'*[*i*].**s-**<formal parameter>.**s-**<sort>)))

endlet)

2.6 Agents

Abstract Syntax

Agent-definition	::	*Agent-name* *Number-of-instances* *Agent-type-identifier*
Number-of-instances	::	*Initial-number* [*Maximum-number*]
Initial-number	=	*NAT*
Maximum-number	=	*NAT*
Agent-formal-parameter	=	*Parameter*
Parameter	::	*Variable-name Sort-reference-identifier*
State-machine-definition	::	*State-name Composite-state-type-identifier*
State-transition-graph	::	[*On-exception*] [*State-start-node*] *State-node***-set** *Free-action***-set** *Exception-handler-node***-set**

Conditions on Abstract Syntax

$\forall d \in$ *Agent-definition*: *d*. *agentKind*$_1$ = **system** ⇒ *d.parentAS1* ∉ *Agent-type-definition*

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

$\forall d \in$ *Agent-definition*: *d*. *agentKind*$_1$ = **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> ::
    [<valid input signal set>]
    <entity in agent>*
    { <interaction> | <agent body> }

<interaction> ::
  { <channel to channel connection>
  | <channel definition>
  | <agent definition>
  | <agent reference>
  | <textual typebased agent definition>}*
  [<state partition>]

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

<agent additional heading> ::

[<specialization>] <formal agent parameter>*

<entity in agent> =
<signal definition>
| <signal reference>
| <signal list definition>
| <variable definition>
| <remote procedure definition>
| <remote variable definition>
| <data definition>
| <data type reference>
| <timer definition>
| <interface reference>
| <exception definition>
| <procedure reference>
| <procedure definitions>
| <composite state type definition>
| <composite state type reference>
| <select definition>
| <agent type definition>
| <agent type reference>
| <gate in definition>

<valid input signal set> :: <signal list item>*

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

<initial number> = <Natural<simple expression>

<maximum number> = <Natural<simple expression>

<formal agent parameter> = <parameters of sort>+

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

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

<agent body gen start> :: [<on exception>] <start>

Conditions on Concrete Syntax

$\forall sp \in$ <state partition>: *sp*. *parentAS0*.*parentAS0* $\in$ <agent structure> $\Rightarrow$
sp.*name*$_0$= *sp*.*parentAS0*.*parentAS0*.*parentAS0* *name*$_0$

The <state partitioning> must have the same name as the containing agent.

$\forall in \in$ <initial number>:$\forall mn \in$ <maximum number>:
in.*parentAS0* = *mn*.*parentAS0* $\Rightarrow$
in.**s-**<simple expression>.*value*$_0 \leq$ *mn*.**s-**<simple expression>.*value*$_0 \wedge$
mn.**s-**<simple expression>.*value*$_0 \geq 0$

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>(
<typebased process heading>(*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 obsolete.

An agent that is a specialization is shorthand for defining an implicit agent type and one typebased agent of this type.

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 11.2.2).

These anonymous variables are accessed using pid expressions as further explained in 12.3.4.3.

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 a shorthand for a list of parameter declarations.

Mapping

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

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

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

Auxiliary Functions

The function $value_0$ computes the Natural value for a simple expression.

$value_0$(e:<simple expression>): *NAT*=$_{def}$
$value_1$((e))

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

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.agentLocalFormalParameterList*$_0$ = *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.

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.

2.6.3 Process

Concrete Syntax

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

<process heading> :: <qualifier> <process<name> <agent instantiation>

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

Procedure-formal-parameter = *In-parameter*
| *Inout-parameter*
| *Out-parameter*
In-parameter :: *Parameter*
Inout-parameter :: *Parameter*
Out-parameter :: *Parameter*

Procedure-graph :: [*On-exception*]
[*Procedure-start-node*]
State-node-**set**
Free-action-**set**
Exception-handler-node-**set**

Result :: *Sort-reference-identifier*

Concrete Syntax

<procedure definitions> =
<external procedure definition>
| <procedure definition>

<procedure definition> ::
<package use clause>* <procedure heading> <entity in procedure>*
{ <procedure body> | <statement>* | <procedure definition gen compound statement> }

<procedure definition gen compound statement> ::
[<virtuality>] <compound statement>

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

<exported> :: [<remote procedure<identifier>]

<procedure heading> ::
<procedure preamble> <qualifier> <procedure<name>
<formal context parameter>*
[<virtuality constraint>]
[<specialization>]
<formal procedure parameter>*
[<procedure result>]
<raises>

<entity in procedure> =
<variable definition>
| <data definition>
| <data type reference>
| <procedure reference>
| <procedure definitions>
| <composite state type definition>
| <composite state type reference>
| <exception definition>
| <select definition>

<procedure body> ::
[<on exception>] [<start>] {<state> | <exception handler> | <free action>}*

<external procedure definition> :: <procedure><name> <procedure signature>

<formal procedure parameter> :: <parameter kind> <parameters of sort>

<parameter kind> = [**inout** | **in** | **out**]

<procedure result> :: [<variable><name>] <sort>

<raises> = <exception><identifier>*

<procedure signature> :: <formal parameter>* [<result>] <raises>

Conditions on Concrete Syntax

$\forall pd \in$ <procedure definitions>:$pd.isExported_0 \Rightarrow$
$pd.formalContextParameterList_0 = empty \wedge$
$pd.surroundingScopeUnit_0 \in$ <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>:
$vd.surroundingScopeUnit_0 \in$ <procedure definitions>$\Rightarrow \neg vd.isExported_0$

<variable definition> in a <procedure definition>, cannot contain **exported** <variable name>s

$\forall pd1, pd2 \in$ <procedure definitions>:
(*parentAS0ofKind*(*pd1*, <agent definition> $\cup$<agent type definition>) =
parentAS0ofKind(*pd2*, <agent definition> $\cup$<agent type definition>) $\wedge$
$pd1.isExported_0 \wedge pd2.isExported_0 \wedge pp1 \neq pp2) \Rightarrow$
(*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>: $te.baseType_0 \notin$ <external procedure definition> $\wedge$
$\forall procCons \in$ <procedure constraint>: *procCons*.**s-**<identifier> $\neq$ *undefined* $\Rightarrow$
getEntityDefinition$_0$(*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 procedure parameter>(*undefined*, *params*) =1=> <formal procedure parameter>(**in**, *params*)

A formal parameter with no explicit <parameter kind> has the implicit <parameter kind> in.

<procedure definition>(*uses*,
h=<procedure heading>(*, *, *, *, *, *, *, *, <procedure result>(*resName*, *resSort*), *), *entities*, *body*)
provided *resName* $\neq$ *undefined* $\wedge$
replaceInSyntaxTree(<return body>(*undefined*), <return body>(*resName*), *body*) $\neq$ *body*
=8=>
<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=<procedure definition>(*uses*,
h=<procedure heading>(*, *, *, *, *, *, *, *, <procedure result>(*resName*, *resSort*), *), *entities*, *body*)
provided *resName* $\neq$ *undefined* $\wedge$
resName $\notin$ { *v*.**s-**<name> | *v* $\in$ <variables of sort gen name>: *v.parentAS0.parentAS0.parentAS0 = p* }

=8=>
<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>, a procedure <start> which contains <virtuality>, or a <statement list> in a <procedure definition> following <virtuality> is called a virtual procedure start. Virtual procedure start is further described in 8.3.3.

< <external procedure definition>(*, *) > =7=> *empty*()

An external procedure definition is not considered in the dynamic semantics.

Mapping to Abstract Syntax

| <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*))
| <procedure definition gen compound statement>(*, *body*) => *Mapping*(*body*))

| <formal procedure parameter>(**in**, *params*) => **mk-***In-parameter*(*Mapping*(*params*))
| <formal procedure parameter>(**out**, *params*) => **mk-***Out-parameter*(*Mapping*(*params*))
| <formal procedure parameter>(**inout**, *params*) => **mk-***Inout-parameter*(*Mapping*(*params*))

Auxiliary Functions

Determine if a <variable definition> or a <procedure definition> is exported.

isExported$_0$(*vp*: <variable definition>∪<procedure definition>): *BOOLEAN*=$_{\text{def}}$
case *vp* **of**
| <variable definition> =>
if *vp*.**s-exported** = *undefined* **then** *False* **else** *True* **endif**
| <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 a <procedure definition>.

procedureFormalParameterList$_0$(*pd*: <procedure definition>): <name>*=$_{\text{def}}$
pd.localProcedureFormalParameterList$_0$ ^ *pd.inheritedProcedureFormalParameterList*$_0$

localProcedureFormalParameterList$_0$(*pd*: <procedure definition>): <name>* =$_{\text{def}}$
let *fpl*=*pd*.**s-**<procedure heading>.**s-**<formal procedure parameter>**-seq in**
if *fpl* = *empty* **then** *empty*
else
< *f*.**s-**<parameters of sort>.**s-**<variable<name>**-seq** | *f* **in** *fpl* >
endif
endlet

inheritedProcedureFormalParameterList$_0$(*pd*: <procedure definition>): <name>* =$_{def}$
let *sp* = *pd.specialization*$_0$ **in**
if *sp* = *undefined* **then** *empty*
else *sp*.**s-**<type expression>*.baseType*$_0$*.procedureFormalParameterList*$_0$
endlet

Get the formal parameter list for a <procedure signature>.

procedureSignatureParameterList$_0$(*ps*: <procedure signature>):<formal parameter>* =$_{def}$
ps.**s-**<formal parameter>**-seq**

Get the formal parameter list for an *Agent-type-definition*, a *Composite-state-type-definition*, or a *Procedure-definition*.

formalParameterList$_1$(*d*: *Agent-type-definition*∪*Composite-state-type-definition*∪*Procedure-definition*):
*Parameter**=$_{def}$
d.localFormalParameterList$_1$^*d.inheritedFormalParameterList*$_1$

localFormalParameterList$_1$(*d*: *Agent-type-definition*∪*Composite-state-type-definition*∪*Procedure-definition*):
*Parameter**=$_{def}$
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*
endcase

inheritedFormalParameterList$_1$(*d*: *Agent-type-definition*∪*Composite-state-type-definition*∪
Procedure-definition): *Parameter**=$_{def}$
let *d'* = *take*({*d'*∈ *Agent-type-definition*∪*Composite-state-type-definition*∪*Procedure-definition*:
isDirectSuperType$_1$(*d'*, *d*)}) **in**
if *d'*≠*undefined* **then** *d'*. *formalParameterList*$_1$
else *empty* **endif**
endlet

Determine if the sort list of *Expression*s corresponds by position to the list of *Sort-reference-identifiers*.

isActualAndFormalParameterMatched$_1$(*expl*:[*Expression*]*, *fpsl*:*Sort-reference-identifier**):*BOOLEAN*=$_{def}$
(*expl.length* = *fpsl.length*)∧
(∀*i*∈ 1..*expl.length*:*expl*[*i*]=*undefined*∨*isCompatibleTo*$_1$(*expr*[*i*].*staticSort*$_1$, *fpsl*[*i*]))

Get the sort list of the formal parameters of the given definition.

formalParameterSortList$_1$(*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*.*formalParameterList*$_1$>
| *Operation-signature*=> <*fa*.**s-***Argument* | *fa* **in** *d*.**s-***Formal-argument*-**seq**>
| *Signal-definition*∪*Timer-definition*∪*Exception-definition*=>*d*.**s-***Sort-reference-identifier*-**seq**
endcase

The function *isSameRaises*$_0$ is used to determine if the two given raises are the same.

isSameRaises$_0$(*raises1*:<raises>, *raises2*:<raises>): *BOOLEAN*=$_{def}$
raises1.**s-**<identifier>. *length*= *raises2*.**s-**<identifier>. *length*∧
(∀*i*∈ 1.. *raises1*.**s-**<identifier>.*length*:
raises1.**s-**<identifier>[*i*] = *raises2*.**s-**<identifier>[*i*])

2.7 Communication

2.7.1 Channel Definition

Abstract Syntax

Channel-definition	::	*Channel-name* [**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

$\forall c \in$ *Channel-definition*: |*c*.**s-***Channel-path*-**set**| $\in \{1, 2\}$

The *Channel-path*-**set** contains at least one *Channel-path* and no more than two.

$\forall\ c \in$ *Channel-definition*: |*c*.**s-***Channel-path*| = 2 $\Rightarrow$
(∀*p*,*p′*∈ *c*.**s-***Channel-path*: *p*≠*p′* $\Rightarrow$
p.**s-***Originating-gate* = *p′*.**s-***Destination-gate* $\wedge$ *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*.

$\forall c \in$ *Channel-definition*: $\forall p \in$ *Channel-path*: $\forall d \in$ *Agent-type-definition*:
$\forall g, g' \in$ *Gate-definition*: (*p.parentAS1*= *c*) $\wedge$ (*g*. *parentAS1*= *d*) $\wedge$ (*g′*. *parentAS1*= *d*) $\wedge$ (*g* ≠ *g′*)$\wedge$
(*p*.**s-***Originating-gate*=*g.identifier*$_1$) $\wedge$ (*p*.**s-***Destination-gate*= *g′.identifier*$_1$) $\Rightarrow$ |*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**).

$\forall c \in$ *Channel-definition*: $\forall p \in$ *Channel-path*: $\exists g, g' \in$ *Gate-definition*:
(*p.parentAS1*= *c*) $\wedge$ (*p*.**s-***Originating-gate*=*g*. *identifier*$_1$) $\wedge$ (*g*. *parentAS1*= *c*. *parentAS1*) $\wedge$
(*p*.**s-***Destination-gate*= *g′.identifier*$_1$) $\wedge$ (*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>] [**nodelay**] <channel path> [<channel path>]

<channel path> :: <channel endpoint> <channel endpoint> <signal list item>*

<channel endpoint> ::
{ <identifier> | **env** | **this**}[<gate>]

Conditions on Concrete Syntax

∀*ce*∈ <channel endpoint>:
let *id* = *ce*.**s-implicit in**
let *g* = *ce*.**s-**<gate> **in**
(*id*∈ <identifier>$\wedge$ *getEntityDefinition*$_0$(*id*,**agent**)∈ <textual typebased agent definition>)$\Rightarrow$
(**let** *d*= *getEntityDefinition*$_0$(*id*, **agent**) **in**
let *td* = *d*.**s-**<type expression>.*baseType*$_0$ **in**
(*ce*.**s-**<gate> ≠ *undefined* $\wedge$
(∃*gd*∈ <textual gate definition>: ∃*gc*∈ <gate constraint>:

gd∈ *td.localGateDefinitionSet*$_0$∧*gc.parentAS0* =*gd*∧
gd.name$_0$=*g*∧ *gc.direction*$_0$ =*ce.direction*$_0$ ∧
gc.**s-**<signal list item>**-seq**.*signalSet*$_0$ ∩
ce.parentAS0.**s-**<signal list item>**-seq**.*signalSet*$_0$≠∅))
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>∧
getEntityDefinition$_0$(*id*, **state**)∈ <textual typebased state partition definition>)⇒
(**let** *d*= *getEntityDefinition*$_0$(*id*, **state**) **in**
let *td*= *d*. **s-**<type expression>.*baseType*$_0$ **in**
(*ce*.**s-**<gate> ≠ *undefined* ∧
(∃*gd*∈ <textual gate definition>: ∃*gc*∈ <gate constraint>:
gd∈ *td.localGateDefinitionSet*$_0$∧*gc.parentAS0* =*gd*∧
gd.name$_0$=*g*∧ *gc.direction*$_0$ =*ce.direction*$_0$ ∧
gc.**s-**<signal list item>**-seq**.*signalSet*$_0$ ∩*ce*.**s-**<signal list item>**-seq**.*signalSet*$_0$≠∅))
endlet)
endlet

<gate> must be specified if <channel endpoint> denotes a connection to a <textual typebased state partition definition> 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.surroundingScopeUnit*$_0$ **in**
(*id* = **env**∧*su*∈ <agent type definition>) ⇒
(∃*gd*∈ <textual gate definition>: ∃*gc*∈ <gate constraint>:
gd∈ *su.localGateDefinitionSet*$_0$∧*gc.parentAS0* =*gd*∧
gd.name$_0$=*g*∧ *gc.direction*$_0$ =*ce.direction*$_0$ ∧
gc.**s-**<signal list item>**-seq**.*signalSet*$_0$ ∩*ce.parentAS0*.**s-**<signal list item>**-seq**.*signalSet*$_0$≠∅)
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

<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.refersto*$_0$*.getEntities.toSet*:
g ∈ <gate in definition> ∧ ¬*isConnected*(*g*, *undefined*, *id.refersto*$_0$*.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 not connected by explicit channels, implicit channels are derived according to the following three steps:

Step 1: Insertion of channels between instance sets inside the agent or agent type and between the instance sets and the agent state machine.

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.

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 modeled 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

SIGNALDIRECTION$_0$ =$_{def}$ { **out**, **in** }

Get the direction of a <gate constraint> or a <channel endpoint>.

direction$_0$(*p*: <gate constraint>∪<channel endpoint>): *SIGNALDIRECTION*$_0$=$_{def}$
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 *origination*$_0$ gets the originating channel endpoint of a <channel path>.

origination$_0$(*p*: <channel path>): <channel endpoint> =$_{def}$
p.**s-**<channel endpoint>

The function *destination*$_0$ gets the destination channel endpoint of a <channel path>.

destination$_0$(*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.

channelEndpointReferTo$_0$(*ep*:<channel endpoint>): *ENTITYDEFINITION*$_0$=$_{def}$
let *end* = *ep*.**s-implicit in**
case *end* **of**
| <identifier>=>*getEntityDefinition*$_0$(*end*, *end.idKind*$_0$)
| **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.identifier*$_0$ **endif**

in
let *id2* =
if *ag2* ∈ <agent type definition> **then env else** *ag2.identifier*$_0$ **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* =$_{\text{def}}$
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>.*refersto*$_0$ ∧
(*g2*=*p*.**s2-**<channel endpoint>.**s-**<gate>.*refersto*$_0$ ∨ *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** =$_{\text{def}}$
U{ *g*.**s-**<signal list item>-**seq** | *g* ∈ <gate constraint>: *g*.**s-implicit** = *kind* }

2.7.2 Connections

Concrete Syntax

<channel to channel connection> ::
<channel><identifier>+ <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.surroundingScopeUnit$_0$= *c2.surroundingScopeUnit*$_0$∧ *c1* ≠ *c2*⇒ *ids1*∩*ids2* = ∅
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.surroundingScopeUnit*$_0$= *c2.surroundingScopeUnit*$_0$∧ *c1* ≠ *c2*)⇒(*ids1* = *ids2* ∨ *ids1*∩*ids2* = ∅)
endlet)

For any pair of <channel to channel connection>s 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*(*undefined*) **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>(*fullQualifier*$_0$(*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* ∧ *fullIdentifier*$_0$(*c*) ∈ *c*.**s2-**<identifier>**-seq**} **in**
matchingGateIds.take
else
let *matchingGateIds* =
{ *c.myImplicitGateIdentifier* | *c* ∈ <channel-to-channel connection>:
c.parentAS0 = *id.refersto*$_0$ ∧ *fullIdentifier*$_0$(*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.refersto*$_0$.**s-**<channel path>.**s-**<signal list item>**-seq** | *id* **in** *c*.**s-**<identifier>**-seq** >) ⌢
BigSeq(< *id.refersto*$_0$.**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.refersto*$_0$.**s-**<channel path>.**s-**<signal list item>**-seq** | *id* **in** *c*.**s-**<identifier>**-seq** >) ⌢
BigSeq(< *id.refersto*$_0$.**s-**<channel path>.**s-**<signal list item>**-seq** | *id* **in** *c*.**s2-**<identifier>**-seq** >)

Mapping

| < <channel to channel connection>(*,*) > => *empty*

2.7.3 Signal

Abstract Syntax

Signal-definition :: *Signal-name Sort-reference-identifier**

Concrete Syntax

<signal definition>:: <type preamble> <signal definition item>+

<signal definition item> ::
<signal<name> <formal context parameter>* [<virtuality constraint>] [<specialization>] <sort>*

Conditions on Concrete Syntax

∀*sdi*∈ <signal definition item>:∀*fcp*∈ <formal context parameter>:
fcp∈ *sdi.localFormalContextParameterList*$_0$*.toSet*⇒ *fcp*∈ <sort context parameter>

In <signal definition>, <formal context parameter> in <formal context parameters> must be a <sort context parameter>.

∀*sdi*∈ <signal definition item>:*sdi.specialization*$_0$.**s-**<type expression>.**s-**<base type>.*idKind*$_0$ = **signal**

The <base type> as part of <specialization> must be a <signal identifier>.

∀*sid*∈ <identifier>:
sid.parentAS0∉ <type expression>∧ *sid.parentAS0*∉ <signal constraint>⇒
¬*getEntityDefinition*$_0$(*sid*, **signal**).*isAbstractType*$_0$

An abstract signal can only be used in specialization and signal constraints.

Transformations

< <signal definition>(*pre*, < *item* > ⌢ *rest*) > **provided** *rest* ≠ *undefined*
=1=> < <signal definition>(*pre*, < *item* >), <signal definition>(*pre*, *rest*) >

If several <signal definition item>s are specified in one signal definition, this is equivalent to individual <signal definition>s for each of them.

Mapping to Abstract Syntax

| <signal definition>(*, < *item* >) => *Mapping*(*item*)
| <signal definition item>(*name*,*,*,*,*sortlist*)
=> **mk-***Signal-definition*(*Mapping*(*name*), **if** *sortlist= undefined* **then** *empty* **else** *Mapping*(*sortlist*) **endif**)

2.7.4 Signal List Definition

Concrete Syntax

<signal list definition> ::
<signal list><name> <signal list item>+

<signal list item> ::
[**signal** | **signallist** | **timer** | **remote procedure** | **interface** | **remote variable**] <identifier>

Conditions on Concrete Syntax

∀*siglistDef*∈ <signal list definition>: ¬*isSiglistContaining*$_0$(*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 item>(*kind*, *id*) > **provided** *id.refersto*$_0$ ∈ <signal list definition>
=8=> *id.refersto*$_0$.**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 definition>(*,*) > => *empty*

| < <signal list item>(*, *id*) > => *Mapping*(*id*)

Auxiliary Functions

The function *isSiglistContaining*$_0$ is used to determine if a signal list contains another signal list, either directly or indirectly.

isSiglistContaining$_0$(*sld1*: <signal list definition>, *sld2*: <signal list definition>): *BOOLEAN* =$_{def}$
∃*sid*∈ <identifier>: *sid.parentAS0.parentAS0*= *sld1*∧*sid.idKind*$_0$ = **signallist**∧
(*getEntityDefinition*$_0$(*sid*, **signallist**) = *sld2*∨
(∃ *sld3*∈ <signal list definition>: *isSiglistContaining*$_0$(*sld1*, *sld3*)∧ *isSiglistContaining*$_0$(*sld3*, *sld2*)))

2.7.5 Remote Procedures

Concrete Syntax

<remote procedure definition> ::
<remote procedure><name> [**nodelay**] <procedure signature>

<remote procedure call> :: <remote procedure call body>

<remote procedure call body> ::
<remote procedure><identifier> [<actual parameter>]* <communication constraint>* [<on exception>]

Conditions on Concrete Syntax

∀*pd*∈ <procedure definitions>:
let *rpi* = *pd*.**s-**<procedure heading>. **s-**<procedure preamble>.**s-**<exported>.**s-**<identifier> **in**
let *rpd* =*getEntityDefinition*$_0$(*rpi*, **remote procedure**) **in**
pd.isExported$_0$ ∧ *rpi*≠ *undefined*⇒
isSameProcedureAndSignature$_0$(*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 definitions>:
let *rpi* = *pd*.**s-**<procedure heading>. **s-**<procedure preamble>.**s-**<exported>.**s-**<identifier> **in**
let *rpd* =*getEntityDefinition*$_0$(*pd.name*$_0$, **remote procedure**) **in**
pd.isExported$_0$ ∧ *rpi* = *undefined*⇒
(*rpd* ≠ *undefined*∧*isSameProcedureAndSignature*$_0$(*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.validOutputSignalSet*$_0$
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.staticSort*$_0$ ≠ "Pid" ⇒
(**let** *def* = *getEntityDefinition*$_0$(*exp.staticSort*$_0$, **sort**) **in**
let *pd* = *getEntityDefinition*$_0$(*callBody*.**s-**<identifier>, **remote procedure**) **in**
def ∈ <interface definition> ∧ *isDefinedIn*$_0$(*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.

∀*id*∈ <identifier>:
id∈ <communication constraint> ⇒*getEntityDefinition*$_0$(*id*, **exception**) = *undefined*

The <timer identifier> of <communication constraints> must not have the same <identifier> as an <exception identifier>.

$\forall c \in$ <output body> :
($\forall d1,d2 \in$ <destination>: $d1.parentAS0 = d2.parentAS0 \wedge c = d1.parentAS0 \Rightarrow d1 = d2$) $\wedge$
($\forall id1,id2 \in$ <identifier>: $id1.parentAS0 = id2.parentAS0 \wedge c = id1.parentAS0 \Rightarrow id1 = id2$)

A <communication constraints> shall contain no more than one <destination> and no more than one <timer identifier>.

Transformations

A remote procedure call

call Proc(apar) to destination timer timerlist 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=> (**let** *esigs* = < <signal definition>(*undefined*,
<signal definition item>(*ex*.**s-**<name> ^ "RAISE", *empty*, *undefined*, *undefined*,
ex.refersto$_0$.**s-**<sort>**-seq** ^ < "Integer" >)) |
ex **in** *sign*.**s-**<raises> > **in**
let *eh* = <exception handler>(< *nn* ^ "EXC" >, *undefined*,
< <handle>(*undefined*, < <exception stimulus>(ex, params) >, *undefined*,
<transition gen action statement>(<
<action statement>(*undefined*,
<output>(<output body>(<
<output body gen identifier>(
<identifier>(*ex*.**s-**<qualifier>, *ex*.**s-**<name> ^ "RAISE"), params) >),
undefined)) >,
<terminator statement>(*undefined*,
<raise>(<raise body>(ex, params), *undefined*)))) |
ex **in** *sign*.**s-**<raises> >) **in**
< *r*,
<signal definition>(*undefined*,
< <signal definition item>(*nn* ^ "CALL", *empty*, *undefined*, *undefined*, < "Integer" >) >),
<signal definition>(*undefined*,
< <signal definition item>(*nn* ^ "REPLY", *empty*, *undefined*, *undefined*, < "Integer" >) >) >
^ *eigs* ^ < *eh* >
endlet)
and
r.implicitName := *nn*

There are two implicit <signal definition>s for each <remote procedure definition>s in a <system definition>. The <signal name>s in these <signal definition>s 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.

For every exception contained in the <raises> of a remote procedure p and all predefined exceptions e a signal eRAISE is defined which can transport all exception parameters of e.

For every exception e contained in the <raises> of the remote procedure, and for every predefined exception the following is inserted:

exceptionhandler pEXC;

handle e(params);

output eRAISE(params,n) to ivar;

raise e(params);

If an exception handler is associated to a remote-procedure input, the exception handler becomes associated to the resulting signal input (not shown in the model above).

timerdef = <timer definition>(< <timer definition item>(*tn*, *sorts*, *) >)
provided ∃ *tid* ∈ <identifier> ∧ *tid.refersto*$_0$ = *timerdef* ∧ *tid.parentAS0* ∈ <remote procedure call body> ∧
¬ *timerdef.hasImplicitexception*
=16=> < *timerdef*,
<exception definition>(< <exception definition item>(*tn*, *sorts*) >)>
and
timerdef.hasImplicitexception:= *True*

For a timer t included in <communication constraints> an additional exception with the same name and the same parameters is implicitly inserted in the same scope as the timer definition, and there must not be an explicitly defined exception with the same name as the timer in the same scope unit where the timer is defined.

< <channel definition>(*n*, *delay*,
<channel path>(*ep1*, *ep2*,
sigs1 ⌢ <signal list item>(**remote procedure**, *i*=<identifier>(*q*, *n*)) ⌢ *sigs2*),
path2, *n2*) >
provided *i.refersto*$_0$*.implicitName* ≠ *undefined*
=17=>
< <channel definition>(*n*, *delay*,
<channel path>(*ep1*, *ep2*, *sigs1* ⌢ < <identifier>(*q*, *i.refersto*$_0$*.implicitName* ⌢ "CALL") > ⌢ *sigs2*),
path2, *n2*),
<channel definition>(*newName*,
<channel path>(*ep2*, *ep1*, < <identifier>(*q*, *i.refersto*$_0$*.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, *undefined*,
<terminator statement>(<raise>(<raise body>(*tid*, *params*), *undefined*))
) >) | *tid* in *constr*: *tid* ∈ <identifier> >
in
let *procDef* = <procedure definition gen compound statement>(*empty*,
<procedure heading>(<procedure preamble>(*undefined*, *undefined*), *undefined*, *nn*,
undefined, *undefined*, *undefined*, *undefined*, *undefined*, *undefined*, *undefined*),
empty,
<procedure body>(*undefined*,
<start>(*undefined*, *undefined*, *undefined*,
<transition gen action statement>(<

```
                    <action statement>(undefined,
                        <task>(<assignment>(varN, <operator application>("+", < varN, "1" >))),
                        undefined),
                    <action statement>(undefined,
                        <output>(<output body>(
                            <output body gen identifier>(id.refersto0.implicitName⌢"CALL", params⌢varN),
                            constr)),
                        undefined),
                >),
                <terminator statement>(undefined,
                    <nextstate>(<nextstate body gen name>("WAIT", undefined, undefined))
                )
        )), <
            <state>(< "Wait" >, undefined, <
                    <save part>(undefined, <asterisk>),
                    <input part>(undefined,
                        < <stimulus>( id.refersto0.implicitName⌢"REPLY", inoutpars ⌢ varNewN) >,
                        undefined, undefined,
                        <transition>(
                            <decision>(<operator application>("=", < varNewN, varN >),
                              <answer>("True", empty)
                              <answer>("False", <terminator statement>(<nextstate>("WAIT")))
                            ),
                            <terminator statement>(<return>(undefined))
                        ) ) >),
                    <input part>(undefined,
                        < <stimulus>( ex.s-<name> ⌢ "RAISE", params ⌢ varNewN) >,
                        undefined, undefined,
                        <transition>(
                            <decision>(<operator application>("=", < varNewN, varN >),
                              <answer>("True", empty)
                              <answer>("False", <terminator statement>(<nextstate>("WAIT")))
                            ),
                            <terminator statement>(<raise>(<raise body>(ex, params), 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, undefined,
                    <terminator statement>(undefined, <dash nextstate>()))
                 <input part>(undefined, < "RAISE" >, undefined, undefined,
                    <terminator statement>(undefined, <dash nextstate>()))>)
        >
```

a) For each imported procedure, two implicit Integer variables n and newn are defined, and n is initialized to 0.

NOTE 1 – The parameter n is introduced to recognize and discard reply signals of remote procedure calls which were left through associated timer expiry.

The <remote procedure call> is transformed as below.

task n:= n + 1;

output pCALL(apar,n) to destination via viapath;

wait in state pWAIT, saving all other signals;

input pREPLY(aINOUTpar,newn);

decision newn = n;

(true):

(false): nextstate pWAIT;

enddecision;

return;

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.

NOTE 2 – The return statement terminates the implicit procedure introduced according to 11.12.1.

The following will be inserted into the state pWAIT:

state pWAIT;

input eRAISE(params,newn);

decision newn = n;

(true): raise e(params);

(false): nextstate pWAIT;

enddecision;

Additionally, the following will be inserted for a timer t that is included in <communication constraints>

state pWait;

input t(aParams);

raise t(aParams);

where aParams stands for implicitly defined variables with the sort of the parameters contained in the timer definition.

In all states of the agent except pWAIT

input pReply, eRAISE;

nextstate actual state;

is inserted.

i=<input part>(*rpc*, *trans*) **provided** *rpc.refersto*$_0$ ∈ <remote procedure definition>
=17=> <input part>(*rpc.refersto*$_0$*.implicitName* ^ "CALL", *fpar* ^ *n*, <transition>(<
<task>(<assignment>(*ivar*, <expression gen primary>(*undefined*, <sender expression>())), *undefined*),
if *rpc.refersto*$_0$.**s-**<procedure signature>.**s-**<result> ≠ *undefined* **then**
<procedure call>(<procedure call body>(*undefined*, *rpc*, *fpar*), *rpc.refersto*$_0$*.implicitName* ^ "EXC")
else
<task>(<assignment>(*res*,
<value returning procedure call>(<procedure call body>(*undefined*, *rpc*, *fpar*))),
rpc.refersto$_0$*.implicitName* ^ "EXC")
endif >
^ *trans*.**s-**<action statement>**-seq**, *trans*.**s-**<terminator statement>))
))
and
states=*i.parentAS0.parentAS0*.**s-**<state>**-seq**
=>
< **if** *handled*(*rpc*, *s*.**s-**<state list>.*head.name*$_0$, *states*) **then** *s*
else <state>(*s*.**s-**<state list>, *s*.**s-**<on exception>, *s*.**s-implicit** ^
<input part>(*rpc.refersto*$_0$*.implicitName* ^ "CALL", *fpar* ^ *n*, <transition>(<
<task>(<assignment>(*ivar*,
<expression gen primary>(*undefined*, <sender expression>())), *undefined*),
if *rpc.refersto*$_0$.**s-**<procedure signature>.**s-**<result> ≠ *undefined* **then**
<procedure call>(
<procedure call body>(*undefined*, *rpc*, *fpar*), *rpc.refersto*$_0$*.implicitName* ^ "EXC")
else

<task>(<assignment>(*res*,
<value returning procedure call>(<procedure call body>(*undefined*, *rpc*, *fpar*))),
rpc.refersto$_0$*.implicitName* ∩ "EXC")
endif >, <terminator statement>(*undefined*, <dash nextstate>()))
))) | *s* **in** *states* >

b) In the server agent, an implicit exception handler pEXC and an implicit Integer variable n is defined for each explicit or implicit <input part> being a remote-procedure input. Furthermore, there is one ivar variable for each such <input part> defined in the scope where the explicit or implicit remote procedure input occurs. If a value returning remote procedure call is transformed, an implicit variable res with the same sort as <sort> in <procedure result> is defined.

To all <state>s with a remote procedure input transition, the following <input part> is added:

input pCALL(fpar,n);

task ivar:= sender;

call Proc(fpar); onexception pEXC;

output pREPLY(INOUTpar,n) to ivar;

transition;

or,

input pCALL(fpar,n);

task ivar:= sender;

task res := call Proc(fpar); onexception pEXC;

output pREPLY(INOUTpar,res,n) to ivar;

transition;

if a value returning remote procedure call was transformed.

To all other <state>s excluding implicit states derived from input, the following <input part> is added:

input pCALL(fpar,n);

task ivar:= sender;

call Proc(fpar); onexception pEXC;

output pREPLY(INOUTpar,n) to ivar;

/* next state the same */

<save part>(*virt*, < <signal list item>(**remote procedure**, *id*) >)
=17=> <save part>(*virt*, < *id.refersto*$_0$*.implicitName* ∩ "CALL" >)

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

save pCall;

<input part>(*virt*, < <stimulus>(<signal list item>(**remote procedure**, *id*),
<remote procedure reject>(<raise body>(*exid*, *params*))) >,
undefined, *undefined*, *trans*)
=17=> <input part>(*virt*, < *id.refersto*$_0$*.implicitName* ∩ "CALL" >, *undefined*, *undefined*,
<transition>(<output>(*exid.name*$_0$ ∩ "RAISE", *params* ∩ *varN*, **sender**) ∩ *trans*)
)

To all <state>s with a <remote procedure reject>, the following <input part> is added:

input pCALL;

output eRAISE(params,n) to sender;

transition;

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

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 *hasImplicitexception* is used to store if already an implicit exception for a timer definition was generated.

controlled *hasImplicitexception*: <timer definition> → *BOOLEAN*

The function *predefExcDefined* is used to store if already the implicit signals for the predefined exceptions were generated.

controlled *predefExcDefined*: → *BOOLEAN*

The function *getEntities* is used to get the entity definitions of the enclosing scope unit.

getEntities(*n*: *DefinitionAS0*): *ENTITYDEFINITION*$_0$* =$_{def}$
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*
| <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*.*name*$_0$ | *si* ∈ *s*.**s-**<state list>.*toSet* } ∧
(*rpc* ∈ **U**{ { *si*.**s-**<signal list item>.**s-**<identifier> | *si* ∈ *input*.**s-**<input list>.*toSet* }
| *input* ∈ *s*.**s-implicit**: *s* ∈ <input part> })
rpc ∈ **U**{ { *si*.**s-**<identifier> | *si* ∈ *save*.**s-**<save list>.*toSet* }
| *save* ∈ *s*.**s-implicit**: *s* ∈ <save part> })
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*
| <procedure definition>(*, *, *entities*, *) => *entities*
otherwise *n.parentAS0.getEntities*
endcase
endif

Determine if a <procedure definition> and a <procedure signature> are matching.

isSameProcedureAndSignature$_0$(*pd*: <procedure definition>, *ps*: <procedure signature>):*BOOLEAN* =$_{def}$
let *fpl* = *ps.procedureSignatureParameterList*$_0$ **in**
let *fpl'* = *pd.procedureFormalParameterList*$_0$ **in**
(*fpl.length* = *fpl'.length*) ∧
(∀*i*∈ 1..*fpl.length*:
(*fpl*[*i*].**s-**<parameter kind> = *fpl'*[*i*].*parentAS0*. *parentAS0*.**s-**<parameter kind>) ∧
isSameSort$_0$(*fpl*[*i*].**s-**<sort>, *fpl'*[*i*].*parentAS0*.**s-**<sort>))∧
isSameResult$_0$(*pd*.**s-**<procedure heading>.**s-**<procedure result>, *ps*.**s-**<result>)∧
isSameRaises$_0$(*pd*.**s-**<procedure heading>.**s-**<raises>, *ps*.**s-**<raises>)
endlet

Determine if two results are matching.

isSameResult$_0$(*r*: <result>∪<procedure result>∪<operation result>,
r′: <result>∪<procedure result>∪<operation result>): *BOOLEAN* =$_{\text{def}}$
isSameSort$_0$(*r*.**s-**<sort>,*r′*.**s-**<sort>)

2.7.6 Remote Variables

Concrete Syntax

<remote variable definition> :: <remote variable definition gen name>+

<remote variable definition gen name> ::
<remote variable<name>+ <sort> [**nodelay**]

<import expression> :: <remote variable<identifier> <communication constraint>*

<export> :: <export body> [<on exception>]

<export body> :: <variable<identifier>+

Conditions on Concrete Syntax

∀*v* ∈ <variables of sort gen name>:
let *rvd* = *getEntityDefinition*$_0$(*v*.**s-**<identifier>, **remote variable**) **in**
v.isExported$_0$∧ *v*.**s-**<identifier> ≠*undefined*⇒*isSameSort*$_0$(*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* =*getEntityDefinition*$_0$(*v*.**s-**<name>, **remote variable**) **in**
v.isExported$_0$∧*v*.**s-**<identifier> = *undefined*⇒
(*rvd* = *undefined* ∧ *isSameSort*$_0$(*v.parentAS0*.**s-**<sort>, *rvd*.**s-**<sort>))
endlet

In case of no **as** clause the remote variable definition in the nearest enclosing scope unit with the same name and sort as the exported variable definition is denoted.

∀*exp*∈ <import expression>: *exp*.**s-**<identifier>∈
(**let** *d=parentAS0ofKind*(*exp*, <agent definition> ∪<agent type definition>) **in**
exp.**s-**<identifier>∈ *d.validOutputSignalSet*$_0$
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>⇒*getEntityDefinition*$_0$(*vid*,**variable**).*isExported*$_0$

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

∀*exp*∈ <expression>: ∀*importExp*∈ <import expression>:
exp. *parentAS0* = *importExp* ∧ *exp.staticSort*$_0$ ≠ "Pid" ⇒
(**let** *def* = *getEntityDefinition*$_0$(*exp.staticSort*$_0$, **sort**) **in**
let *pd* = *getEntityDefinition*$_0$(*importExp*.**s-**<identifier>, **remote variable**) **in**
def ∈ <interface definition> ∧ *isDefinedIn*$_0$(*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 variable definition gen name>(< *n* >, *sort*, *delay*) >) >
provided *r.implicitName = undefined*
=16=> < *r*,
<signal definition>(*undefined*,
< <signal definition item>(*n* ∩ "QUERY", *empty*, *undefined*, *undefined*, < "Integer" >) >),
<signal definition>(*undefined*,
< <signal definition item>(*n* ∩ "REPLY", *empty*, *undefined*, *undefined*,
< *sort*, "Integer" >) >) >
and
r.implicitName := *nn*

There are two implicit <signal definition>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 $i.refersto_0.implicitName \neq undefined$
=16=>
< <channel definition>(*n*, *delay*,
<channel path>(*ep1*, *ep2*, *sigs1* ∩ < <identifier>(*q*, *i*.**s-**<name> ∩ "QUERY") > ∩ *sigs2*),
path2, *n2*),
<channel definition>(*newName*,
<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 case of a channel, the new channel has the same delaying property as the original one.

let *entities*=
if *rootNodeAS0*.s-<textual system specification> ≠ *undefined*
then *rootNodeAS0*.s-<textual system specification>*.getEntities*
else *rootNodeAS0*.s-<package definition>*.head.getEntities*
endif

```
in
entities provided ¬ predefExcDefined
=16=>
    entities ⌢
    < <exception definition>(< <exception definition item>(ex.s-<name> ⌢ "RAISE", empty) >)
    | ex in predefexceptions >
and
    predefExcDefined:= True
```

For each predefined exception (denoted as predefExc), an additional anonymous signal (denoted as predefExcRAISE) is defined.

```
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, undefined,
            <terminator statement>(<raise>(<raise body>(tid, params), undefined))
            ) >) | tid in constr: tid ∈ <identifier> >
    in
    let procDef = <procedure definition gen compound statement>(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 gen action statement>(<
                        <action statement>(undefined,
                            <task>(<assignment>(varN, <operator application>("+", < varN, "1" >))),
                            undefined),
                        <action statement>(undefined,
                            <output>(<output body>(
                                <output body gen identifier>(id.refersto0.implicitName ⌢ "QUERY",
                                  params ⌢ varN), constr)),
                            undefined),
                    >),
                    <terminator statement>(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, undefined,
                            <transition>(
                                <decision>(<operator application>("=", < varNewN, varN >),
                                  <answer>("True", empty)
```

```
                                <answer>("False", <terminator statement>(<nextstate>("WAIT")))
                            ),
                            <terminator statement>(<return>(undefined))
                        ) ) >),
                    <input part>(undefined,
                        < <stimulus>( ex.s-<name> ^ "RAISE", params ^ varNewN) >,
                        undefined, undefined,
                        <transition>(
                            <decision>(<operator application>("=", < varNewN, varN >),
                              <answer>("True", empty)
                              <answer>("False", <terminator statement>(<nextstate>("WAIT")))
                            ),
                            <terminator statement>(<raise>(<raise body>(ex, params), 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, undefined,
                  <terminator statement>(undefined, <dash nextstate>()))
                <input part>(undefined, < "RAISE" >, undefined, undefined,
                  <terminator statement>(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:

task n:= n + 1;

output xQUERY(n) to destination via via-path;

wait in state xWAIT, saving all other signals;

input xREPLY(x,newn);

decision newn = n;

(true):

(false): nextstate xWAIT;

enddecision;

return;

state xWAIT

input predefExcRAISE;

raise predefExc;

In all other states, xREPLY is saved.

NOTE 1 – The return statement terminates the implicit procedure introduced according to 11.12.1.

For every timer t included in <communication constraints>, an additional exception with the same name and the same parameters is implicitly inserted in the same scope as the timer definition. In that case there must not be an exception with the same name in the scope unit of the timer definition.

Additionally, the following will be inserted for every timer t that is included in <communication constraints>:

state xWait;

input t(aParams);

raise t(aParams);

where aParams stand for implicitly defined variables with the sort of the parameters contained in the timer definition.

The result of the transformation is encapsulated in an implicit procedure, as described in 11.12.1. Every <on exception> attached to the import action shall be attached to a call of the implicit procedure.

i=<input part>(*rv*, *trans*) **provided** *rpc.refersto*$_0$ ∈ <remote variable definition>
=17=> <input part>(*rv*.s-<name> ⌢ "QUERY", *n*, <transition>(<
<task>(<assignment>(*ivar*, <expression gen primary>(*undefined*, <sender expression>())), *undefined*),
<output>(<output body>(<
<output body gen identifier>(*rv*.s-<name> ⌢ "QUERY", *rpc.refersto*$_0$*.implicitName*) >,
< <destination>(*ivar*)>),
rv.s-<name> ⌢ "QUERY")
>
⌢ *trans*.**s-**<action statement>**-seq**, *trans*.**s-**<terminator statement>))
))

b) *Exporter*

To all <state>s of the exporter, excluding implicit states derived from import, the following <input part> is added:

input xQUERY(n);

task ivar := sender;

output xREPLY(imcx,n) to ivar; onexception xEXC;

nextstate the state containing this input;

exceptionhandler xEXC;

handle predefExc;

output predefExcRAISE to ivar;

raise predefExc;

For each such state, ivar will be defined as variable of sort Pid, and n as a variable of type Integer.

<export>(<export body>(< *id* >), *exc*)
=17=>
<task>(<assignment>(*id.refersto*$_0$*.implicitName*, *id*), *exc*)

The <export>

export x

is transformed to the following:

task imcx := x;

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

2.8 Behaviour

2.8.1 Start

Abstract Syntax

State-start-node :: [*On-exception*] [*State-entry-point-name*] *Transition*
Procedure-start-node :: [*On-exception*] *Transition*

Concrete Syntax

<start> :: [<virtuality>] [<state entry point<name>] [<on exception>] <transition>

Conditions on Concrete Syntax

∀*s*∈ <start>:
(*s*.**s-**<name> ≠ *undefined*) ⇒
(*s*.*surroundingScopeUnit*$_0$∈ <composite state>)∨
(*s*.*surroundingScopeUnit*$_0$∈ <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*,*exc*,*trans*)
=> **if** *s*.*parentAS0* ∈ <procedure body> **then mk-***Procedure-start-node*(*Mapping*(*exc*), *Mapping*(*trans*))
else mk-*State-start-node*(*Mapping*(*exc*), *Mapping*(*entry*), *Mapping*(*trans*)) **endif**

2.8.2 State

Abstract Syntax

State-node	::	*State-name* [*On-exception*] *Save-signalset* *Input-node***-set** *Spontaneous-transition***-set** *Continuous-signal***-set** *Connect-node***-set** [*Composite-state-type-identifier*]

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*.

∀*d*∈ *TYPEDEFINITION*$_1$:∀*in*,*in′*∈ *Input-node*:
(*in*≠*in′*∧*in*.*parentAS1*.**s-***State-name*=*in′*.*parentAS1*.**s-***State-name*∧
in.*parentAS1*∈ *d*.*stateNodeSet*$_1$∧*in′*.*parentAS1*∈ *d*.*stateNodeSet*$_1$)⇒
in.**s-***Signal-identifier*≠*in′*.**s-***Signal-identifier*

The *Signal-identifier*s in the *Input-node***-set** must be distinct.

Concrete Syntax

<state> ::
<state list> [<on exception>]
{ <input part>
| <priority input>
| <save part>
| <spontaneous transition>
| <continuous signal>
| <connect part>}*

<state list> = {<state list item> | <typebased composite state>}+ | <asterisk state list>

<state list item> :: <state<name> [<actual parameter>]*

<asterisk state list> ::<state<name>*

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> ($sn.parentAS0 = sn'.parentAS0$) $\Rightarrow$ ($sn \neq sn'$)) $\wedge$
($sn \in bs.surroundingScopeUnit_0.stateNameSet_0$)

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> :
($r.referencedDefinition_0 = r'.referencedDefinition_0$) $\Rightarrow$
($r.parentAS0 = r'.parentAS0$)$\wedge$
($\exists!csa \in$ <state>:$\exists sn \in$ <name>:
($sn \in csa.stateNameSet_0$)$\wedge$ //*sn* is a state name of *csa*.
($sn = r.surroundingScopeUnit_0.entityName_0$)) //the 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*, *exc*, *triggers*) > **provided** *rest* $\neq$ *empty*
=1=> < <state>(< *s* >, *exc*, *triggers*), <state>(*rest*, *exc*, *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>s contain the same <state name>, these <state>s are concatenated into one <state> having that <state name>.

b=<state>(<asterisk state list>(*exceptStates*), *exc*, *triggers*)
=13=>
<state>(< *s* **in** *sort*(*b.surroundingScopeUnit*$_0$*.stateNameSet*$_0$): *s* $\notin$ *exceptStates.toSet* >, *exc*, *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>(< <state list item>(*name*, *undefined*) >, *exc*, *triggers*)
=> **mk-***State-node*(*Mapping*(*name*), *Mapping*(*exc*),
{ *t* $\in$ *Mapping*(*triggers*).*toSet*: *t* $\in$ *Save-signalset*},
{ *t* $\in$ *Mapping*(*triggers*).*toSet*: *t* $\in$ *Input-node*},
{ *t* $\in$ *Mapping*(*triggers*).*toSet*: *t* $\in$ *Spontaneous-transition*},
{ *t* $\in$ *Mapping*(*triggers*).*toSet*: *t* $\in$ *Continuous-signal*}, *undefined*)

| <state>(< <typebased composite state>(*name*, *parent*) >, *exc*, *triggers*)
=> **mk-***State-node*(*Mapping*(*name*), *Mapping*(*exc*),
{ *t* $\in$ *Mapping*(*triggers*).*toSet*: *t* $\in$ *Save-signalset*},
{ *t* $\in$ *Mapping*(*triggers*).*toSet*: *t* $\in$ *Input-node*},
{ *t* $\in$ *Mapping*(*triggers*).*toSet*: *t* $\in$ *Spontaneous-transition*},
{ *t* $\in$ *Mapping*(*triggers*).*toSet*: *t* $\in$ *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.

stateNameSet$_0$(*d*: <state>$\cup$<agent definition> $\cup$<agent type definition> $\cup$<composite state> $\cup$
<composite state type definition>$\cup$<procedure definitions>): <name>**-set** $=_{def}$

if (*d*∈ <state>) **then**
  if *d*.**s**-<state list>∈ <asterisk state list>**then**
    $d.surroundingScopeUnit_0.stateNameSet_0$\{*n*∈ <name>: *n.parentAS0*=*d*.**s**-<state list>}
  else // *d*.**s**-<state list>∈ {<state list item> | <typebased composite state>}+
    {*n*∈ <name>: *n.parentAS0*=*d*.**s**-<state list>}
  endif
else {*n*∈ <name>: ∃*s*∈ <state>: (*s*∈ $d.stateSet_0$)∧(*n*∈ $s.stateNameSet_0$)}
endif

2.8.3 Input

Abstract Syntax

Input-node :: [**PRIORITY**]
Signal-identifier
[*Variable-identifier*]*
[*Provided-expression*]
[*On-exception*]
Transition

Conditions on Abstract Syntax

∀*in* ∈ *Input-node*: ∀*sd* ∈ *Signal-definition*:
  sd= $getEntityDefinition_1$(*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*=$getEntityDefinition_1$(*in*.**s**-*Variable-identifier*[*i*],**variable**)∧
        $isCompatibleTo_1$(*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 *Sort-reference-identifiers* 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> [<on exception>] [<provided expression>] <transition>

<input list> = <stimulus>+ | <asterisk input list>

<stimulus> :: <signal list item> [[<variable>]+ | <remote procedure reject>]

<remote procedure reject> :: <raise body>

<asterisk input list> :: ()

Conditions on Concrete Syntax

∀*s*∈ <state>: |$s.asteriskInputListSet_0$| ≤ 1

A <state> may contain at most one <asterisk input list>.

∀*s*∈ <state>: (*s*.$asteriskInputListSet_0$=∅) ∨ (*s*.$asteriskSaveListSet_0$ = ∅)

A <state> must not contain both <asterisk input list> and <asterisk save list>.

∀*ip*∈ <input part>: ∀*rpr*∈ <remote procedure reject>:
  isAncestorAS0(*ip*,*rpr*)⇒*rpr.parentAS0*.**s**-<signal list item>.**s**-<identifier>.$idKind_0$= **remote procedure** ∧
    (**let** *rpd* = $getEntityDefinition_0$(*rpr.parentAS0*.**s**-<signal list item>.**s**-<identifier>,**remote procedure**) **in**
      rpr.**s**-<raise body>.**s**-<identifier>∈ *toSet*(*rpd*.**s**-<procedure signature>.**s**-<raises> **endlet**)

A <remote procedure reject> may be specified only if the <signal list item> denotes a <remote procedure identifier>. The <exception identifier> in the <remote procedure reject> must be mentioned in the <remote procedure definition>.

∀*ip*∈ <input part>: ∀*sli*∈ <signal list item>:
isAncestorAS0(*ip*,*sli*)⇒
(**let** *idKind* = *sli*.**s-**<identifier>.*idKind*$_0$**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.refersto*$_0$.**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*, *exc*, *cond*, *trans*) > **provided** *rest* ≠ *empty*
=1=> < <input part>(*virt*, < *stim* >, *exc*, *cond*, *trans*), <input part>(*virt*, *rest*, *exc*, *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.

<input part>(*virt*, < <stimulus>(*item*, *vars*) >, *exc*, *cond*, <transition gen action statement>(*actions*, *term*))
provided *vars* ∉ <remote procedure reject> ∧
{ *v* ∈ *vars.toSet*: *v* ∈ (<indexed variable> ∪ <field variable>) } ≠ ∅
=8=>
(**let** *newvars* =
< (**if** *v* ∈ (<indexed variable> ∪ <field variable>) **then** *newName*(*v*) **else** *v* **endif**) | *v* **in** *vars* > **in**
let *newtrans* = <transition gen action statement>(
< <action statement>(*undefined*, <task>(<assignment>(*v*, *newName*(*v*)), *undefined*))
| *v* **in** *vars*: (*v* ∈ (<indexed variable> ∪ <field variable>))>
⌢ *actions*, *term*) **in**
<input part>(*virt*, < <stimulus>(*item*, *newvars*) >, *exc*, *cond*, *newtrans*)
endlet)

When one or more of the <variable>s of a <stimulus> are <indexed variable>s or <field variable>s, then all the <variable>s are replaced by unique, new, implicitly declared <variable identifier>s. Directly following the <input part>, a <task> is inserted which in its <textual task body> contains an <assignment> for each of the <variable>s, assigning the result of the corresponding new variable to the <variable>. The results will be assigned in the order from left to right of the list of <variable>s. This <task> becomes the first <action statement> in the <transition>.

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 10.5, 10.6, 11.4, 11.5, and 11.6 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*) >, *exc*, *cond*, *trans*)
=> **mk-***Input-node*(*undefined*, *Mapping*(*item*),
Mapping(*vars*), *Mapping*(*cond*), *Mapping*(*exc*), *Mapping*(*trans*))

Auxiliary Functions

Get the <asterisk input list> for a <state>.

asteriskInputListSet$_0$(*s*: <state>): <asterisk input list>**-set** =$_{def}$
{*ail*∈ <asterisk input list>: *isAncestorAS0*(*s*,*ail*)}

2.8.4 Priority Input

Concrete Syntax

<priority input> ::
[<virtuality>] <priority input list> [<on exception>]<transition>

<priority input list> = <stimulus>+

Mapping to Abstract Syntax

| <priority input>(*, <priority input list>(< <stimulus>(*item*, *vars*) >), *exc*, *trans*)
=> **mk-***Input-node*(**PRIORITY**, *Mapping*(*item*), *Mapping*(*vars*), *undefined*, *Mapping*(*exc*), *Mapping*(*trans*))

2.8.5 Continuous Signal

Abstract Syntax

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

Concrete Syntax

<continuous signal> ::
[<virtuality>] <continuous expression> [<priority name>] [<on exception>] <transition>
<continuous expression> = <Boolean<expression>
<priority name> = <Natural<name>

Mapping to Abstract Syntax

| <continuous signal>(*, *expr*, *prio*, *exc*, *trans*)
=> **mk-***Continuous-signal*(*Mapping*(*expr*), *Mapping*(*prio*), *Mapping*(*trans*))

2.8.6 Enabling Condition

Abstract Syntax

Provided-expression	=	*Boolean-expression*
Boolean-expression	=	*Expression*

Concrete Syntax

<provided expression>:: <Boolean<expression>

Transformations

let *nn*=*newName*() **in**
p=<provided expression>(*expr*)
provided ∃ *i* ∈ <imperative expression>: *isAncestorAS0*(*i*, *p*)
=8=>
<provided expression>(<value returning procedure call>(<procedure call body>(*undefined*, *nn*, *empty*)))
and
entities = *p.surroundingScopeUnit*$_0$*.getEntities*
=> *entities* ∩
<procedure definition>(*empty*,
<procedure heading>(<procedure preamble>(*undefined*, *undefined*), *empty*, *nn*, *empty*, *undefined*,
undefined, *empty*, <result>("Boolean"), *empty*),

empty,
<procedure body>(*undefined*,
<start>(*undefined*, *undefined*, *undefined*,
<terminator statement>(*undefined*, <return>(<return body>(*expr*), *undefined*))),
empty))

When the <provided expression> contains an <imperative expression>, the following procedure with an anonymous name referred to as isEnabled is implicitly defined.

procedure isEnabled -> Boolean;

start;

return <provided expression>;

endprocedure;

NOTE – The <Boolean expression> may be further transformed according to the model of <import expression>.

Mapping to Abstract Syntax

| <provided expression>(*expr*) => *Mapping*(*expr*)

2.8.7 Save

Abstract Syntax

Save-signalset = *Signal-identifier*-**set**

Concrete Syntax

<save part> :: [<virtuality>] <save list>

<save list> = <signal list item>+ | <asterisk save list>

<asterisk save list> :: ()

Conditions on Concrete Syntax

$\forall s \in$ <state>: $|s.asteriskSaveListSet_0| \leq 1$

A <state> may contain at most one <asterisk save list>.

$\forall s \in$ <state>: $(s.asteriskInputListSet_0 = \varnothing) \vee (s.asteriskSaveListSet_0 = \varnothing)$

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 10.5, 10.6, 11.4, 11.5, and 11.6 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>.

asteriskSaveListSet$_0$(*s*: <state>): <asterisk save list>-**set** $=_{def}$
{*asl*$\in$ <asterisk save list>: *isAncestorAS0*(*s*,*asl*)}

2.8.8 Implicit Transition

Transformations

The following statement is handled by the dynamic semantics.

For each <state> there is an implicit <input part> containing a <transition> that only contains a <nextstate> leading back to the same <state>.

2.8.9 Spontaneous Transition

Abstract Syntax

Spontaneous-transition :: [*On-exception*]
[*Provided-expression*]
Transition

Concrete Syntax

<spontaneous transition> ::
[<virtuality>] [<on exception>] [<provided expression>] <transition>

Mapping to Abstract Syntax

| <spontaneous transition>(*, *exc*, *cond*, *trans*)
=> **mk-***Spontaneous-transition*(*Mapping*(*exc*), *Mapping*(*cond*), *Mapping*(*trans*))

2.8.10 Label

Abstract Syntax

Free-action :: *Connector-name Transition*

Concrete Syntax

<label> :: <connector<name>

<free action> :: <transition>

Conditions on Concrete Syntax

$\forall b \in$ <composite state body>$\cup$<operation body>$\cup$<procedure body>$\cup$<agent body>: $\forall l, l' \in$ <label>:
($isAncestorAS0(b,l) \wedge isAncestorAS0(b,l') \wedge (l \neq l') \Rightarrow (l$.**s-**<name> $\neq l'$.**s-**<name>))

All the <connector name>s defined in a body must be distinct.

$\forall fa \in$ <free action>: $\forall l \in$ <label>:
(($l = fa$.**s-**<transition>.**s-**<terminator statement>.**s-**<label>) $\vee$
($l = fa$.**s-**<transition>.**s-**<action statement>**-seq**.*head*.**s-**<label>)) $\Rightarrow$
(l.**s-**<name> = *fa*.**s-**<name>)

If the <transition string> of the <transition> in <free action> is non-empty, the first <action statement> must have a <label> otherwise the <terminator statement> 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 statement> = *empty* **then** *t*.**s-**<terminator statement>.**s-**<label>
else *t*.**s-**<action statement>.*head*.**s-**<label>
endif

Transformations

< *a*, *s*=<action statement>(*l*, *) > ⌢ *r* **provided** *l* ≠ *undefined*
=5=> < *a* >
and
a.parentAS0 => <transition gen action statement>(*a.parentAS0*.**s-**<action statement>,
<terminator statement>(*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 statement>s preceding the <label> are preserved in the original transition, which is terminated with a <join> to the <label>. All action statements 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*))

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* [*On-exception*] *Transition*

Conditions on Abstract Syntax

(∀*d* ∈ *Entry-procedure-definition*:
(*d*.**s-***Procedure-name*= "entry")∧(*d.formalParameterList*$_1$ = *empty*)∧(*d.stateNodeSet*$_1$=∅))

(∀*d* ∈ *Exit-procedure-definition*:
(*d*.**s-***Procedure-name*="exit")∧(*d.formalParameterList*$_1$ = *empty*)∧(*d*. *stateNodeSet*$_1$=∅))

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> ::
<package use clause>* <composite state heading> <composite state structure>

<composite state structure> ::
[<valid input signal set>] <gate in definition>*
<state connection points>* <entity in composite state>*
{ <composite state body> | <state aggregation body> }

<composite state heading> ::
<qualifier> <composite state<name> <formal agent parameter>*

<entity in composite state> =
<variable definition>
| <data definition>
| <select definition>

| <data type reference>
| <procedure definitions>
| <exception definition>
| <composite state type definition>
| <composite state type reference>

2.8.11.1 Composite State Graph

Abstract Syntax

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

Concrete Syntax

<composite state body> ::
[<on exception>] <start>* {<state> | <exception handler> | <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>: $pn \in csd.usedEntryNameSet_0 \Rightarrow pn \in csd.definedEntryNameSet_0$)$\wedge$
($\forall pn \in$ <name>: $pn \in csd.usedExitNameSet_0 \Rightarrow pn \in csd.definedExitNameSet_0$)

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>: ($s \in csb.surroundingScopeUnit_0.stateSet_0$) $\wedge$
((s.**s**-<state list>$\notin$ <asterisk state list>)) $\Rightarrow$ $csg.surroundingScopeUnit_0.startSet_0 \neq \varnothing$

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>:
($vd.surroundingScopeUnit_0 = cs$) $\wedge$ ($cs.surroundingScopeUnit_0 \in$ <procedure definitions>) $\Rightarrow$
(vd.**s-exported** = *undefined*)

<variable definition> in a <composite state>, cannot contain **exported** <variable name>s, if the <composite state> is enclosed by a <procedure definition>.

Transformations

<composite state body>(*exc*, *empty*,
items1 $\frown$ <state>(<asterisk state list>(*undefined*), *exc*, *triggers*, *undefined*) $\frown$ *items2*)
provided < *i* in (*items1* $\frown$ *items2*): (*i* $\in$ <state>) > = *empty*
=8=>
let *nn* = *newName* **in**
let *startTrans* = <transition gen action statement>(*empty*,
<terminator statement>(*undefined*,
<nextstate>(<nextstate body gen name>(*nn*, *undefined*, *undefined*)))))) **in**
<composite state body>(*exc*, < <start>(*undefined*, *undefined*, *undefined*, *startTrans*) >,
items1 $\frown$ <state>(< *nn* >, exc, *triggers*, *undefined*) $\frown$ *items2*)
endlet)

<composite state body>(*exc*, *empty*,
i1 $\frown$ <state>(<asterisk state list>(*undefined*), *exc*, *triggers*, *undefined*) $\frown$ *i2*)
provided < *i* in (*items1* $\frown$ *items2*): (*i* $\in$ <state>) > = *empty*
=8=>

(**let** *nn* = *newName* **in**
let *startTrans* = <transition gen action statement>(*empty*,
<terminator statement>(*undefined*,
<nextstate>(<nextstate body gen name>(*nn*, *undefined*, *undefined*)))) **in**
<composite state body>(*exc*, < <start>(*undefined*, *undefined*, *undefined*, *startTrans*) >,
i1 ⌢ <state>(< *nn* >, *exc*, *triggers*, *undefined*) ⌢ *i2*)
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>(*exc*, *starts*, *items*)))
=> **mk-***Composite-state-graph*(
mk-*State-transition-graph*(*Mapping*(*exc*), *head*(< *s* **in** *Mapping*(*starts*): (*s* ∈ *State-start-node*)>),
{ *s* ∈ *Mapping*(*items*): *s* ∈ *State-node* },
{ *s* ∈ *Mapping*(*items*): *s* ∈ *Free-action*},
{ *s* ∈ *Mapping*(*items*): *s* ∈ *Exception-handler-node* }),
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 name used in a <composite state> or a <composite state type definition>.

$usedEntryNameSet_0$(*csd*: <composite state>∪<composite state type definition>):<name>-**set**=$_{def}$
{*n*∈ <name>: *n.parentAS0*∈ *csd.*$startSet_0$}

Get the set of exit name used in a <composite state> or a <composite state type definition>.

$usedExitNameSet_0$(*csd*: <composite state>∪<composite state type definition>):<name>-**set**=$_{def}$
{*n*∈ <name>: ∃*s*∈ *csd.*$stateSet_0$: *isAncestorAS0*(*s*, *n*)∧(*n.parentAS0.parentAS0*∈ <return>)}

Get the set of entry name defined in a <composite state> or a <composite state type definition>.

$definedEntryNameSet_0$(*csd*: <composite state>∪<composite state type definition>):<name>-**set**=$_{def}$
{*n*∈ <name>: (*n.parentAS0*∈ <state entry points>)∧
(*n.parentAS0.parentAS0*=*csd*.**s-**<composite state structure>)}

Get the set of exit name defined in a <composite state> or a <composite state type definition>.

$definedExitNameSet_0$(*csd*: <composite state>∪<composite state type definition>):<name>-**set**=$_{def}$
{*n*∈ <name>: (*n.parentAS0*∈ <state exit points>)∧
(*n.parentAS0.parentAS0*=*csd*.**s-**<composite state structure>)}

2.8.11.2 State Aggregation

Abstract Syntax

State-aggregation-node	::	*State-partition** [*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* | **DEFAULT** }
Inner-entry-point :: { *State-entry-point-name* | **DEFAULT** }
Exit-connection-definition :: *Outer-exit-point Inner-exit-point*
Outer-exit-point :: { *State-exit-point-name* | **DEFAULT** }
Inner-exit-point :: { *State-exit-point-name* | **DEFAULT** }

Conditions on Abstract Syntax

$\forall pn \in$ *State-entry-point-name*: (*pn.parentAS1* $\in$ *Outer-entry-point*) $\Rightarrow$
(*pn* $\in$ *pn.surroundingScopeUnit*$_0$.*entryPointSet*$_1$) $\wedge$
(*pn.surroundingScopeUnit*$_0$.**s-implicit** $\in$ *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.

$\forall pn \in$ *State-entry-point-name*: (*pn.parentAS1* $\in$ *Inner-entry-point*) $\Rightarrow$
(*pn* $\in$ *parentAS1ofKind*(*pn*, *State-partition*).*baseType*$_1$.*entryPointSet*$_1$)

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*.

$\forall pn \in$ *State-exit-point-name*: (*pn.parentAS1* $\in$ *Outer-exit-point*) $\Rightarrow$
(*pn* $\in$ *pn.surroundingScopeUnit*$_0$.*exitPointSet*$_1$) $\wedge$
(*pn.surroundingScopeUnit*$_0$.**s-implicit** $\in$ *State-aggregation-node*)

$\forall pn \in$ *State-exit-point-name*: (*pn.parentAS1* $\in$ *Inner-exit-point*) $\Rightarrow$
(*pn* $\in$ *parentAS1ofKind*(*pn*, *State-partition*).*baseType*$_1$.*exitPointSet*$_1$)

Likewise, the *Outer-exit-point*s must denote exit points in the inner and outer composite state, respectively.

$\forall td \in$ *Composite-state-type-definition*: (*td*.**s-implicit** $\in$ *State-aggregation-node*) $\Rightarrow$
$\exists! cd \in$ *Connection-definition*: (*cd.surroundingScopeUnit*$_0$ = *td*) $\wedge$
(**let** *pointSet* = {*pn* $\in$ *State-entry-point-definition* $\cup$ *State-exit-point-definition*:
(*pn* $\in$ *td.entryPointSet*$_1$ $\cup$ *td.exitPointSet*$_1$) $\vee$
($\exists sp \in$ *State-partition*: (*sp*. *surroundingScopeUnit*$_0$ = *td*) $\wedge$
(*pn* $\in$ *sp.baseType*$_1$.*entryPointSet*$_1$ $\cup$ *sp.baseType*$_1$.*exitPointSet*$_1$))} **in**
let *pointSet'* = {*pn* $\in$ *State-entry-point-definition* $\cup$ *State-exit-point-definition*: *isAncestorAS1*(*cd*,*pn*)} **in**
pointSet $\subseteq$ *pointSet'*
endlet)

All entry and exit points of the both the container state and the state partitions must appear in exactly one *Connection-definition*.

$\forall sp, sp' \in$ *State-partition*: ($sp \neq sp'$) $\wedge$ (*sp.parentAS1* = *sp'. parentAS1*) $\Rightarrow$
sp.inputSignalSet$_1$ $\cap$ *sp'. inputSignalSet*$_1$ = $\emptyset$

The input signal sets of the *State-partition*s within a composite state must be disjoint.

Concrete Syntax

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

<state partition connection> = <state partition connection gen entry> | <state partition connection gen exit>

<state partition connection gen entry> :: <entry point> <entry point>

<state partition connection gen exit> :: <exit point> <exit point>

<state partition> =
<textual typebased state partition definition> | <composite state reference> | <composite state>

<entry point> :: <composite state><identifier> { <state entry point<name> | **default** }

<exit point> :: <composite state><identifier> { <state exit point<name> | **default** }

Transformations

<composite state structure>(*inset*, *gates*, *p* = * ⌢ <state entry points>(* ⌢ < *n* > ⌢ *) ⌢ *,
entities, *body* = * ⌢ *s* ⌢ *)
provided *s* ∈ <state partition> ∧
< *i* in *body*: (*i* ∈ <state partition connection gen entry> ∧ *i*.**s-**<entry point>.**s-implicit** = *n* ∧
i.**s-**<entry point>.**s-**<identifier> = *s*.*identifier*$_0$) > = *empty*
=8=>
<composite state structure>(*inset*, *gates*, *p*, *entities*,
<state partition connection gen entry>(<entry point>(*undefined*, *n*),
<entry point>(*s*.*identifier*$_0$, **default**)) ⌢ *body*)

<composite state structure>(*inset*, *gates*, *p* = * ⌢ <state exit points>(* ⌢ < *n* > ⌢ *) ⌢ *,
entities, *body* = * ⌢ *s* ⌢ *)
provided *s* ∈ <state partition> ∧
< *i* in *body*: (*i* ∈ <state partition connection gen exit> ∧ *i*.**s-**<exit point>.**s-implicit** = *n* ∧
i.**s-**<exit point>.**s-**<identifier> = *s*.*identifier*$_0$) > = *empty*
=8=>
<composite state structure>(*inset*, *gates*, *p*, *entities*,
<state partition connection gen exit>(<exit point>(*undefined*, *n*),
<exit point>(*s*.*identifier*$_0$, **default**)) ⌢ *body*)

If an entry point of the state aggregation is not connected to any entry point of a state partition, an implicit connection to the unlabelled entry is added. Likewise, if an exit point of a partition is not connected to any exit point of the state aggregation, a connection to the unlabelled exit is added.

The following statement is formalised by the dynamic semantics.

If there are signals in the complete valid input set of an agent which 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 appeared in a type definition, a state partition or a state node.

inputSignalSet$_1$(*sp*: *TYPEDEFINITION*$_1$∪*State-partition* ∪*State-node*): *SIGNAL*$_1$ =$_{\text{def}}$
if *sp*∈ *State-node* **then**
sp.**s-***Save-signalset*∪{*in*.**s-***Signal-identifier*: *in*∈*sp*.**s-***Input-node*-**set**}∪
{*getEntityDefinition*$_1$(*cn*.**s-***Procedure-identifier*,**procedure**).*inputSignalSet*$_1$:
cn∈ {*cn*∈ *Call-node*: *isAncestorAS1*(*sp*, *cn*)}}
else // *sp*∈ *TYPEDEFINITION*$_1$∪*State-partition*
{*sn*.*inputSignalSet*$_1$: *sn*∈*sp*.*stateNodeSet*$_1$}
endif

Get the base type of a definition.

baseType$_1$(*as*: *Agent-definition*∪*State-machine-definition*∪*State-partition*∪*State-node*):

Agent-type-definition∪*Composite-state-type-definition*=$_{\text{def}}$
case *as* **of**
| *Agent-definition* => *getEntityDefinition*$_1$(*as*.**s-***Agent-type-identifier* **agent type**)
| *State-machine-definition*∪*State-partition*=>
getEntityDefinition$_1$ (*as*.**s-***Composite-state-type-identifier*, **state type**)
| *State-node* =>
if *as*.**s-***Composite-state-type-identifier*=*undefined* **then** *undefined*
else *getEntityDefinition*$_1$ (*as*.**s-***Composite-state-type-identifier*, **state type**)
otherwise *undefined*
endcase

Get the set of state nodes included in a type definition, state node or a state partition.

stateNodeSet$_1$(*d*: *TYPEDEFINITION*$_1$∪*State-node*∪*State-partition*): *State-node*-**set** =$_{\text{def}}$
case *d* **of**
| *TYPEDEFINITION*$_1$=>*d.localStateNodeSet*$_1$∪*d.inheritedStateNodeSet*$_1$
| *State-node*=>
if *d*.**s-***Composite-state-type-identifier* ≠ *undefined* **then** *d.baseType*$_1$.*stateNodeSet*$_1$∪{*d*}
else {*d*}
| *State-partition*=> *d.baseType*$_1$.*stateNodeSet*$_1$
otherwise=>∅
endcase

Get the set of state nodes defined locally in a type definition.

localStateNodeSet$_1$(*d*: *TYPEDEFINITION*$_1$): *State-node*-**set** =$_{\text{def}}$
case *d* **of**
| *Agent-type-definition*=>
if *d*.*State-machine-definition* ≠ *undefined* **then**
d.*State-machine-definition.baseType*$_1$.*stateNodeSet*$_1$
else ∅
| *Procedure-definition*=>
{*sn.stateNodeSet*$_1$: *sn*∈ *d*.**s-***Procedure-graph*.**s-***State-node*-**set**}
| *Composite-state-type-definition*=>
if *d*.**s-implicit**∈ *Composite-state-graph* **then**
{*sn.stateNodeSet*$_1$: *sn*∈ *d*.**s-implicit**.**s-***State-transition-graph*.**s-***State-node*-**set**}
else // *d*.**s-implicit**∈ *State-aggregation-node*
{*sp.stateNodeSet*$_1$: *sp*∈ *d*.**s-implicit**.**s-***State-partition*-**seq**.*toSet*}
otherwise=>∅
endcase

Get the set of state nodes defined in a super type.

inheritedStateNodeSet$_1$(*d*: *TYPEDEFINITION*$_1$): *State-node*-**set** =$_{\text{def}}$
case *d* **of**
| *Agent-type-definition*=>
if *d*.**s-***Agent-type-identifier*≠ *undefined* **then**
getEntityDefinition$_1$(*d*.**s-***Agent-type-identifier*, **agent type**).*stateNodeSet*$_1$
else ∅
| *Procedure-definition*=>
if *d*.**s-***Procedure-identifier*≠ *undefined* **then**
getEntityDefinition$_1$(*d*.**s-***Procedure-identifier*, **procedure**).*stateNodeSet*$_1$
else ∅
| *Composite-state-type-definition*=>
if *d*.**s-***Composite-state-type-identifier* ≠ *undefined* **then**
getEntityDefinition$_1$(*d*.**s-***Composite-state-type-identifier*, **state type**).*stateNodeSet*$_1$
else ∅
otherwise=>∅
endcase

Get the set of the state entry points of a *Composite-state-type-definition*.

entryPointSet$_1$(*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*.

exitPointSet$_1$(*d*: *Composite-state-type-definition*): *Name*-**set**=$_{def}$
d.**s-***State-exit-point-definition*-**set**

Mapping to Abstract Syntax

| <state partition connection gen entry>(<entry point>(*, *n1*), <entry point>(*, *n2*))
=> **mk-***Entry-connection-definition*(**mk-***Outer-entry-point*(*Mapping*(*n1*), **mk-***Inner-entry-point*(*Mapping*(*n2*))

| <state partition connection gen exit>(<exit point>(*, *n1*), <exit point>(*, *n2*))
=> **mk-***Exit-connection-definition*(**mk-***Outer-exit-point*(*Mapping*(*n1*), **mk-***Inner-exit-point*(*Mapping*(*n2*))

2.8.11.3 State Connection Point

Concrete Syntax

<state connection points> = <state entry points> | <state exit points>

<state entry points> :: <state entry point<name>+

<state exit points> :: <state exit point<name>+

Mapping to Abstract Syntax

| <state entry points>(*x*) => *Mapping*(*x*)

| <state exit points>(*x*) => *Mapping*(*x*)

2.8.11.4 Connect

Abstract Syntax

Connect-node :: [*State-exit-point-name*] [*On-exception*] *Transition*

Concrete Syntax

<connect part> ::
[<virtuality>] <connect list> [<on exception>] <transition>

<connect list> = <state exit point<name>* | <asterisk connect list>

<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*, *exc*, *trans*) > **provided** *rest* $\neq$ *empty*
=1=> < <connect part>(*virt*, < *n* >, *exc*, *trans*), <connect part>(*virt*, *rest*, *exc*, *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>, *exc*, *trans*)
=13=>
(**let** *parentType* = *c.parentAS0*.**s-**<state list>.*head*.**s-**<type expression>.*refersto*$_0$ **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*, *exc*, *trans*)
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* >, *exc*, *trans*) =>
mk-*Connect-node*(*Mapping*(*name*), *Mapping*(*exc*), *Mapping*(*trans*))

2.8.12 Transition

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*
}
[*On-exception*]

Terminator ::
{ *Nextstate-node*
| *Stop-node*
| *Return-node*
| *Join-node*
| *Continue-node*
| *Break-node*
| *Raise-node*
}
[*On-exception*]

Concrete Syntax

<transition> = <transition gen action statement> | <terminator statement>

<transition gen action statement> :: <action statement>+ [<terminator statement>]

<action statement> :: [<label>] <action>
<action> =
<task>
| <output>
| <create request>
| <decision>
| <set>

| <reset>
| <export>
| <procedure call>
| <remote procedure call>
| <transition option>

<terminator statement> :: [<label>] <terminator>

<terminator> = <nextstate> | <join> | <stop> | <return> | <raise>

Conditions on Concrete Syntax

$\forall t \in$ <transition>: ($t \notin$ <terminator statement>)$\wedge$(t.**s-**<terminator statement> = *undefined*)$\wedge$
(t.*parentAS0*.*parentAS0*.*parentAS0* $\notin$ <decision>$\cup$<transition option>) $\Rightarrow$
(**let** *asl* = t.**s-**<action statement>**-seq in**
(*asl*.*last*.**s-**<action> $\in$ <decision>$\cup$<transition option>)$\wedge$
isTransitionTerminating$_0$(*asl*.*last*.<action>) **endlet**)

endlet)

If the <terminator statement> 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 statement>(*,*) **provided** t.*parentAS0* $\notin$ <transition>
=1=> <transition gen action statement>(*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>. To preserve an exception handler associated with the original action, terminator, or decision, this list of actions is encapsulated in a new, implicitly defined procedure with an anonymous name, here referred to as Actions, as follows, where list-of-actions refers to the resultant list of actions:

procedure Actions;

start;

list-of-actions;

return;

endprocedure;

The old action is replaced by a call to Actions. If an exception handler was associated with the original action, the exception handler is associated with the call to Actions.

If the transformed construct occurred in a terminator or decision, the original terminator or decision is replaced by a call to Actions, followed by the new terminator or decision. If an exception handler was associated with the original terminator or decision, the exception handler is associated with the call to Actions and with the new terminator or decision.

No exception handler is associated with the body of Actions or with any part of this body.

Mapping to Abstract Syntax

| <transition gen action statement>(s, t)
=> **if** t = *undefined* **then mk-***Transition*(*Mapping*(< x **in** s: ($x \notin$ <decision>) >), *Mapping*(*s*.*last*))
else mk-*Transition*(*Mapping*(*s*), *Mapping*(*t*))
endif

| <action statement>(*, a) => *Mapping*(a)

Auxiliary Functions

Determine if a <decision> or a <transition option> is terminating.

isTransitionTerminating$_0$(*dt*: <decision>∪<transition option>): *BOOLEAN* =$_{\text{def}}$
∀*t*∈ <transition>: (*t.parentAS0.parentAS0.parentAS0*= *dt*) ⇒
((*t*∈ <terminator statement>)∨
((*t*∉ <terminator statement>)∧(*t*.**s**-<terminator statement> ≠ *undefined*))∨
((*t*∉ <terminator statement>)∧(*t*.**s**-<terminator statement> = *undefined*)∧
isTransitionTerminating$_0$(*t*.**s**-<action statement>-**seq**.*last*.<action>))

2.8.12.2 Nextstate

Abstract Syntax

Nextstate-node	=	*Named-nextstate* | *Dash-nextstate*
Dash-nextstate	::	[**HISTORY**]
Named-nextstate	::	*State-name* [*Nextstate-parameters*]
Nextstate-parameters	::	[*Expression*]*
		[*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*. *Nextstate-parameters* may only be present if *State-name* denotes a composite state.

Concrete Syntax

<nextstate> = <nextstate body>

<nextstate body> = <nextstate body gen name> | <dash nextstate> | <history dash nextstate>

<nextstate body gen name> ::
<state><name> [<actual parameter>]* [<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>: *isReachableFrom*$_0$(*hdn*, *t*)) ⇒
(*s.surroundingScopeUnit*$_0$∈ <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>:∀*n*∈ <dash nextstate>: (*t.parentAS0*∈ <start>) ⇒ ¬*isReachableFrom*$_0$(*n*, *t*)

The <transition> contained in a <start> must not lead, directly or indirectly, to a <dash nextstate>.

∀*t*∈ <transition>: ∀*n*∈ <history dash nextstate>: (*t.parentAS0*∈ <start>∪<handle>) ⇒
¬*isReachableFrom*$_0$(*n*, *t*)

The <transition> contained in a <start> or a <handle> must not lead, directly or indirectly, to a <history dash nextstate>.

∀*oe*∈ <on exception>: ∀*eh*∈ <exception handler>:
((*oe.parentAS0*∈ <start>)∨
(*oe.parentAS0*∈ <composite state body> ∪<agent body>∪<procedure body>∪<operation body>)) ∧
(*eh*∈ *oe.reachableExceptionHandlerSet*$_0$) ⇒
(∀*dn*∈ <dash nextstate>:∀*t*∈ <transition>:*t.parentAS0* **in** *eh*.**s-**<handle>**-seq**⇒
¬*isReachableFrom*$_0$(*dn*, *t*)))

An <on exception> within a <start> or associated to a whole body must not, directly or indirectly (through <on exception>s within <exception handler>s), lead to an <exception handler> containing <dash nextstate>s.

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, priotity inputs, continuous signals, enabling conditions, implicit tasks for imperative actions and remote varibales or procedures.

The rest of this Model section describes how the meaning of <dash nextstate> in exception handlers is determined.

An exception handler is called reachable from a state or exception handler if it is either associated to the state or exception handler, the stimuli attached to the state or exception handler or if it is associated with the transition actions following the stimuli. All exception handlers reachable from an exception handler that is reachable from the state are also called reachable from the state.

NOTE – Reachability is transitive.

For each <state>, the following rule applies: All reachable exception handlers are made distinct for the state by copying each exception handler to an <exception handler> with a new name. The <on exception>s are modified using this new name. Afterwards, exception handlers not reachable from any state are removed.

After this replacement, a given <exception handler> containing <dash nextstate>s can be reached, directly or indirectly, from exactly one <state>. The <dash nextstate>s within each such <exception handler> are replaced by the <state name> of this <state>.

Mapping to Abstract Syntax

| <nextstate body gen name>(*name*, *params*, *entry*)
=> **mk-***Terminator*(**mk-***Named-nextstate*(*Mapping*(*name*),
mk-*Nextstate-parameters*(*Mapping*(*params*), *Mapping*(*entry*))) , *undefined*)

| <dash nextstate>() => **mk-***Terminator*(**mk-***Dash-nextstate*(*undefined*), *undefined*)

| <history dash nextstate>() => **mk-***Terminator*(**mk-***Dash-nextstate*(**HISTORY**), *undefined*)

Auxiliary Functions

The function *exceptionHandlerNameSet*$_0$ gets the set of <exception handler name>s appeared in the given definition.

exceptionHandlerNameSet$_0$(*ed*: <exception handler>∪<agent definition>∪<agent type definition>∪
<composite state>∪<composite state type definition>∪<procedure definitions>): <name>**-set** =$_{def}$
if *ed*∈ <exception handler>**then**
if *ed*.**s-**<exception handler list>∈ <asterisk exception handler list> **then**
ed.surroundingScopeUnit$_0$*.exceptionHandlerNameSet*$_0$\
ed.**s-**<exception handler list>.**s-**<name>**-seq**.*toSet*
else *ed*.**s-**<exception handler list>.*toSet*
endif
else
{*n*∈ <name>: ∃*eh*∈ <exception handler>:
n∈ *eh.exceptionHandlerNameSet*$_0$∧*eh*∈ *ed.exceptionHandlerSet*$_0$}
endif

The function *reachableExceptionHandlerSet0* gets the set of <exception handler>s reachable from an <on exception>.

reachableExceptionHandlerSet$_0$(*oe*: <on exception>): <exception handler>**-set** =$_{def}$
{*eh*∈ <exception handler>:
(*eh* = *oe*.**s**-<name>.*associatedExceptionHandler*$_0$) ∨
(∃*oe′*∈ <on exception>:
isAncestorAS0(*oe*.**s**-<name>.*associatedExceptionHandler*$_0$,*oe′*)∧
(*eh*∈ *oe′*.*reachableExceptionHandlerSet*$_0$))}

The function *isReachableFrom*$_0$ determines if a <nextstate> is reachable from a <transition>.

isReachableFrom$_0$(*n*:<nextstate>, *t*:<transition>):*BOOLEAN*=$_{def}$
isAncestorAS0(*t*, *n*)∨
(∃*j*∈ <join>: *isAncestorAS0*(*t*, *n*)∧
(∃*l*∈ <label>: *j*.**s**-<name> = *l*.**s**-<name>∧
parentAS0ofKind(*l*,<agent body>∪<procedure body>∪<operation body>∪<composite state body>)=
parentAS0ofKind(*j*,<agent body>∪<procedure body>∪<operation body>∪<composite state body>)∧
isReachableFrom$_0$(*n*, *parentAS0ofKind*(*l*, <transition>))))

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*)

2.8.12.4 Stop

Abstract Syntax

Stop-node :: ()

Conditions on Abstract Syntax

∀*sn* ∈ *Stop-node*: ¬(∃*pg* ∈ *Procedure-graph*: *isAncestorAS1*(*pg*, *sn*))

A *Stop-node* must not be contained in a *Procedure-graph*.

Concrete Syntax

<stop>:: ()

Mapping to Abstract Syntax

| <stop>() => **mk-***Terminator*(**mk-***Stop-node*(), *undefined*)

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

$\forall rn \in$ *Return-node*: $\exists pg \in$ *Procedure-graph*: *isAncestorAS1*(*pg*, *rn*)

A *Return-node* must be contained in a *Procedure-graph*.

$\forall rn \in$ *Action-return-node*: $\exists d \in$ *Procedure-definition*:
isAncestorAS1(*d*.**s**-*Procedure-graph*, *rn*) $\wedge$ *d*.**s**-*Result* = *undefined*

An *Action-return-node* must only be contained in the *Procedure-graph* of a *Procedure-definition* without *Result*.

$\forall rn \in$ *Value-return-node*: $\exists d \in$ *Procedure-definition*:
isAncestorAS1(*d*.**s**-*Procedure-graph*, *rn*) $\wedge$ *d*.**s**-*Result* $\neq$ *undefined*

A *Value-return-node* must only be contained in the *Procedure-graph* of a *Procedure-definition* containin *Result*.

$\forall rn \in$ *Named-return-node*: $\exists sg \in$ *Composite-state-graph*: *isAncestorAS1*(*sg*, *rn*)

A *Named-return-node* must only be contained in a *Composite-state-graph*.

Concrete Syntax

<return> :: <return body> [<on exception>]

<return body> :: [<expression> | <state exit point<name>]

Mapping to Abstract Syntax

| <return>(*x*, *exc*)
=> **if** *x* = *undefined* **then mk-***Terminator*(**mk-***Action-return-node*(), *Mapping*(*exc*))
elseif *x* $\in$ <name> **then mk-***Terminator*(**mk-***Named-return-node*(*Mapping*(*x*)), *Mapping*(*exc*))
else mk-*Terminator*(**mk-***Value-return-node*(*Mapping*(*x*)), *Mapping*(*exc*))
endif

2.8.12.6 Raise

Abstract Syntax

Raise-node :: *Exception-identifier* [*Expression*]*

Conditions on Abstract Syntax

$\forall rn \in$ *Raise-node*: $\forall d \in$ *Exception-definition*:
((d=*getEntityDefinition*$_1$(*rn*.**s**-*Exception-identifier*,**exception**)) $\Rightarrow$
(*rn*.**s**-*Expression*-**seq**.*length*= *d*.**s**-*Sort-reference-identifier*-**seq**.*length*)$\wedge$
($\forall i \in$ 1..*rn*.**s**-*Expression*-**seq**.*length*:
rn.**s**-*Expression*[*i*]= *undefined*$\vee$
isCompatibleTo$_1$(*rn*.**s**-*Expression*[*i*].*staticSort*$_1$, *d*.**s**-*Sort-reference-identifier*[*i*])))

In a *Raise-node*, the length of the list of optional *Expression*s must be the same as the number of *Sort-reference-identifiers* in the *Exception-definition* denoted by the *Exception-identifier*. Each *Expression* must have a sort that is compatible to the corresponding (by position) *Sort-reference-identifier* in the *Exception-definition*.

Concrete Syntax

<raise> :: <raise body> [<on exception>]

<raise body> :: <exception><identifier> [<actual parameter>]*

Transformations

The following transformation is handled in the transformations for remote procedure call and import expression.

A raise may be transformed to a list of actions (possibly containing implicit states) plus a new raise according to the model (of remote procedure calls, for example). Then the model for transition terminators in 11.12.1 applies.

Mapping to Abstract Syntax

| <raise>(*r*, *exc*) => **mk-***Terminator*(*Mapping*(*r*), *Mapping*(*exc*))

| <raise body>(*id*, *param*) => **mk-***Raise-node*(*Mapping*(*id*), *Mapping*(*param*))

2.8.13 Action

2.8.13.1 Task

Abstract Syntax

Task-node	=	*Assignment*
	|	*Assignment-attempt*
	|	*Informal-text*

Concrete Syntax

<task> :: { <assignment> | <informal text> | <compound statement> } [<on exception>]

Transformations

< <task>(<compound statement>(<statement list>(*, *empty*))) >
=1=> *empty*

If the <statement list> in the <compound statement> of <textual task body> is empty, then the <task> is removed. If the <statement list> in a <graphical task body> is empty, the <task area> 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.

The following statement is handled by the Mapping of <compound statement>.

A <task> containing a <compound statement> is transformed as shown in 11.14.1. The result of this transformation is inserted in place of <task>.

NOTE – The transform of a <task> or <task area> containing a <statement list> is not necessarily mapped onto a Task-node in the Abstract Grammar.

Mapping to Abstract Syntax

| <task>(*t*, *exc*) => **mk-***Graph-node*(*Mapping*(*t*), *Mapping*(*exc*))

2.8.13.2 Create

Abstract Syntax

Create-request-node	::	{ *Agent-identifier* | **THIS** }
		[*Expression*]*

Conditions on Abstract Syntax

$\forall n \in$ *Create-request-node*: $\forall d \in$ *Agent-definition*:
(*d*=*getEntityDefinition*$_1$(*n*.**s**-*Agent-identifier*,**agent**))$\Rightarrow$

(∃*t* ∈ *Agent-type-definition*:
(*t*=*getEntityDefinition*$_1$(*d*. **s**-*Agent-type-identifier*,**agent type**)) ∧
isActualAndFormalParameterMatched$_1$(*n*.**s**-*Expression*-**seq**, *t*.*formalParameterSortList*$_1$))

The length of the list of optional *Expression*s must be the same as the number of *Agent-formal-parameter*s in the *Agent-definition* of the *Agent-identifier* and each *Expression* 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> [<on exception>]

<create body> :: { <identifier> | **this** } [<actual parameter>]*

<actual parameter> = <expression>

Conditions on Concrete Syntax

∀*cr*∈ <create body>: (*cr*.**s-implicit** = **this**) ⇒
(*cr*.*surroundingScopeUnit*$_0$∈ <agent type definition>) ∧
(*cr*.*surroundingScopeUnit*$_0$.*surroundingScopeUnit*$_0$∈ <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), exc)
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), exc)
        else <decision>(any, exc, <decision body>(<
                <answer part>(undefined,
                    <transition gen action statement>(<action statement>(undefined,
                        <create request>(<create body>(elem.identifier0, params), exc)),
                        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), exc)
provided id.refersto0 ∈ <agent type definition> ∧ c.possibleInstances0 = empty
=8=>
    <create request>(<create body>(<identifier>(undefined, nn), params), exc)
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.refersto*$_0$ ∈ <block type definition>
then <textual typebased block definition>(
<typebased block heading>(*nn*, <number of instances>(*undefined*, *undefined*),
<type expression>(*id*, *empty*)))
else *id.refersto*$_0$ ∈ <process type definition>
then <textual typebased process definition>(
<typebased process heading>(*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.

possibleInstances$_0$(*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*, *exc*) => **mk-***Graph-node*(*Mapping*(*c*), *Mapping*(*exc*))

| <create body>(*id*, *params*) => **mk-***Create-request-node*(*Mapping*(*id*), *Mapping*(*params*))

2.8.13.3 Procedure Call

Abstract Syntax

Call-node :: [**THIS**] *Procedure-identifier* [*Expression*]*

Conditions on Abstract Syntax

∀*n* ∈ *Call-node* ∪ *Value-returning-call-node*: ∀*d* ∈ *Procedure-definition*:
(*d*=*getEntityDefinition*$_1$(*n*. **s**-*Procedure-identifier*,**procedure**)) ⇒
(*isActualAndFormalParameterMatched*$_1$(*n*.**s**-*Expression*-**seq**, *d.formalParameterSortList*$_1$)∧
(∀*i* ∈ 1..*n*.**s**-*Expression*-**seq**.*length*:
d. *formalParameterList*$_1$[*i*] ∈ *Inout-parameter* ∪ *Out-parameter*⇒
n.**s**-*Expression*[*i*]∈ *Identifier*∧*n*.**s**-*Expression*[*i*].*idKind*$_1$= **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* which is sort compatible to the sort identified by the *Sort-reference-identifier* of the *Procedure-formal-parameter*.

Concrete Syntax

<procedure call> :: <procedure call body> [<on exception>]

<procedure call body> ::
[**this**] {procedure<identifier> | procedure<type expression>} [<actual parameter>]*

Conditions on Concrete Syntax

∀*pc*∈ <procedure call>:
(**let** *apl* = *pc*.**s-**<procedure call body>.**s-**<actual parameter>**-seq in**
let *fpl* = *pc.calledProcedure*$_0$.*procedureFormalParameterList*$_0$ **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*, <procedure definition>)= *getEntityDefinition*$_0$(*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.refersto*$_0$, <agent type definition>) ≠
parentAS0ofKind(*p*, <agent type definition>)
=8=>
let *par*=*parentAS0ofKind*(*p*, <agent type definition>) **in**
<procedure call body>(
<identifier>(*par.fullQualifier*$_0$^<path item>(*par.entityKind*$_0$, *par.entityName*$_0$), *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* ^
<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.

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*, *exc*) => **mk-***Graph-node*(*Mapping*(*p*), *Mapping*(*exc*))

| <procedure call body>(*t*, *id*, *params*) => **mk-***Call-node*(*Mapping*(*t*), *Mapping*(*id*), *Mapping*(*params*))

2.8.13.4 Output

Abstract Syntax

Output-node	::	*Signal-identifier* [*Expression*]* [*Signal-destination*] *Direct-via*
Signal-destination	=	*Expression* | *Agent-identifier* | **THIS**
Direct-via	=	(*Channel-identifier* | *Gate-identifier*)**-set**

Conditions on Abstract Syntax

$\forall n \in$ *Output-node*: $\exists d \in$ *Signal-definition*:
(d=*getEntityDefinition*$_1$(n.**s**-*Signal-identifier*,**signal**)) $\wedge$
isActualAndFormalParameterMatched$_1$(n.**s**-*Expression*-**seq**, d.*formalParameterSortList*$_1$)

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.*

$\forall n \in$ *Output-node*: $\forall$ *cid* $\in$ *Channel-identifier*: (*cid*.*parentAS1*$\in$ n.**s**-*Direct-via*) $\Rightarrow$
($\exists d \in$ *Channel-definition*: $\exists p \in$ *Channel-path*:$\exists g \in$ *Gate-definition*:
(d=*getEntityDefinition*$_1$(*cid*,**channel**)) $\wedge$
(p. *parentAS1*=d)$\wedge$
(*parentAS1ofKind*(g, *Agent-type-definition*) =*parentAS1ofKind* (n, *Agent-type-definition*))$\wedge$
(g.*identifier*$_1$ = p.**s**-*Originating-gate*) $\wedge$
(n.**s**-*Signal-identifier* $\in$ g.**s**-*Out-signal-identifier*-**set**)$\wedge$
(n.**s**-*Signal-identifier* $\in$ p.**s**-*Signal-identifier*-**set**))

For each *Channel-identifier* in *Direct-via* there must exist zero or more channels such that the channel via this path is reachable with the *Signal-identifier* from the agent, and the *Channel-path* in the direction from the agent must include *Signal-identifier* in its set of *Signal-identifier*s.

Concrete Syntax

<output> :: <output body> [<on exception>]

<output body> :: <output body gen identifier>+ <communication constraint>*

<output body gen identifier> :: <signal<identifier> [<actual parameter>]*

<communication constraint> =
<destination> | <timer<identifier> | <via path>

<destination> :: { <expression> | <identifier> | **this** }

<via path> :: <identifier>

Conditions on Concrete Syntax

$\forall op \in$ <output>:
(**let** *cc* = *op*.**s**-<output body>.**s**-<communication constraint>-**seq in**
(|{*tid*$\in$ <identifier>: *tid*.*idKind*$_0$= **timer** $\wedge$*tid* **in** *cc*}| = 0) $\wedge$
(|{$d \in$ <destination>: d **in** *cc*}| $\leq$ 1) $\wedge$
(|{*vp*$\in$ <via path>: *vp* **in** *cc*}| $\geq$ 0)
endlet)

The <communication constraints> in an <output> shall contain no **timer <**timer identifier> clause. It contains at most one **to** <destination> clause and zero or more **<**via path>s.

$\forall op \in$ <output>: $\forall dt \in$ <destination>:
(*dt*.*parentAS0* = *op*.**s**-<output body>) $\wedge$ (*dt*.**s-implicit** = **this**) $\Rightarrow$
(*op*.*surroundingScopeUnit*$_0 \in$ <agent type definition>) $\wedge$
(*op*.*surroundingScopeUnit*$_0$.*surroundingScopeUnit*$_0 \in$ <agent type definition>)

this may only be specified in an <agent type definition> and in scopes enclosed by an <agent type definition>.

$\forall op \in$ <output>:$\forall dt \in$ <destination>:
(*dt*.*parentAS0* = *op*.**s**-<output body>) $\wedge$
(*dt*.**s-implicit** $\in$ <expression>) $\wedge$ (*dt*.*staticSort*$_0 \neq$ "Pid") $\Rightarrow$
(**let** *fd* = *getEntityDefinition*$_0$(*dt*.*staticSort*$_0$, **interface**) **in**
$\forall sig \in$ <identifier>: (*sig*.*parentAS0* = *op*.**s**-<output body>) $\Rightarrow$
(*sig*$\in$*fd*.*usedSignalSet*$_0$) $\vee$ (*getEntityDefinition*$_0$(*sig*, **signal**)$\in$*fd*.*definedSignalSet*$_0$) **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.

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*, *exc*) => **mk-***Graph-node*(*Mapping*(*o*), *Mapping*(*exc*))

| <output body>(< <output body gen identifier>(*id*, *params*) >, *constr*)
=> **mk-***Output-node*(*Mapping*(*id*), *Mapping*(*params*),
Mapping(*head*(< *c* **in** *constr*: (*c* ∈ <destination>) >)),
Mapping(< *c* **in** *constr*: (*c* ∈ <via path>) >).*toSet*)

| <destination>(*d*) => *Mapping*(*d*)

Auxiliary Functions

The function $usedSignalSet_0$ gets the set of signals used in an <interface definition>.

$usedSignalSet_0$(*fd*: <interface definition>): $SIGNAL_0$ =$_{def}$
{*sig*∈ $SIGNAL_0$:
(*sig*∈*fd*.**s-**<interface use list>.**s-**<signal list item>**-seq**.$signalSet_0$) ∨
(∃*fd′*∈ <interface definition>: $isSubtype_0$(*fd*, *fd′*) ∧
(*sig*∈*fd′*.**s-**<interface use list>.**s-**<signal list item>**-seq**.$signalSet_0$))}

The function $definedSignalSet_0$ gets the set of signals defined in an <interface definition>.

$definedSignalSet_0$(*fd*: <interface definition>): <signal definition item>**-set** =$_{def}$
{*sd*∈ <signal definition item>:
((*sd*.*parentAS0*∈ <signal definition>)∧(*sd*.*parentAS0*.*parentAS0*∈ <entity in interface>) ∧
(*sd*.*parentAS0*.*parentAS0*.*parentAS0* = *fd*)) ∨
(∃*fd′*∈ <interface definition>: $isSubtype_0$(*fd*, *fd′*) ∧ (*sd*.*parentAS0*∈ <signal definition>) ∧
(*sd*.*parentAS0*.*parentAS0*∈ <entity in interface>) ∧
(*sd*.*parentAS0*.*parentAS0*.*parentAS0*= *fd′*))}

2.8.13.5 Decision

Abstract Syntax

Decision-node	::	*Decision-question* [*On-exception*] *Decision-answer***-set** [*Else-answer*]
Decision-question	=	*Expression* | *Informal-text*
Decision-answer	::	{ *Range-condition* | *Informal-text* } *Transition*
Else-answer	::	*Transition*

Conditions on Abstract Syntax

$\forall dn \in$ *Decision-node*: $\forall r, r' \in$ *Range-condition*: $\forall ce,ce' \in$ *Constant-expression*:
isAncestorAS1(r ,*ce*)$\wedge$ *isAncestorAS1*(r',*ce'*) $\wedge$ ($ce \neq ce'$)$\wedge$
r'. *parentAS1*$\in$ *dn*.**s**-*Decision-answer*-**set** $\wedge$ *r*. *parentAS1*$\in$ *dn*.**s**-*Decision-answer*-**set**$\Rightarrow$
(*isCompatibleTo*$_1$(*ce*. *staticSort*$_1$, *ce'*. *staticSort*$_1$)$\vee$ *isCompatibleTo*$_1$(*ce'*. *staticSort*$_1$, *ce*. *staticSort*$_1$))$\wedge$
(*dn*.**s**-*Decision-question*$\in$ *Expression*$\Rightarrow$
isCompatibleTo$_1$(*ce*. *staticSort*$_1$, *dn*.**s**-*Decision-question*. *staticSort*$_1$))

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.

$\forall dn \in$ *Decision-node*: $\forall r, r' \in$ *Range-condition*:
r'. *parentAS1* $\in$ *dn*.**s**-*Decision-answer*-**set** $\wedge$ *r*. *parentAS1*$\in$ *dn*.**s**-*Decision-answer*-**set** $\wedge$
r. *parentAS1*$\neq$ *r'*. *parentAS1* $\Rightarrow$ *r.range1* $\cap$ *r'*. *range1* = $\varnothing$

The *Range-condition*s of the *Decision-answer*s must be mutually exclusive.

Concrete Syntax

<decision> :: <question> [<on exception>] <decision body>

<decision body> :: <answer part>+ [<else part>]

<answer part> :: [<answer>] [<transition>]

<answer> = <range condition> | <informal text>

<else part> :: [<transition>]

<question> = <expression> | <informal text> | **any**

Conditions on Concrete Syntax

$\forall d \in$ <decision>: (*d*.**s**-<question> = **any**) $\Leftrightarrow$
$\neg$($\exists ap \in$ <answer part>: (*ap.parentAS0.parentAS0*= *d*)) $\wedge$ (*ap*.**s**-<answer> $\neq$ *undefined*) $\wedge$
(*d*.**s**-<decision body>.**s**-<else part> = *undefined*)

The <answer> of <answer part> must be omitted if and only if the <question> consists of the keyword **any**. In this case, no <else part> may be present.

Transformations

<else part>(*undefined*) =1=> <else part>(<transition gen action statement>(*empty*, *undefined*))
<answer part>(*a*, *undefined*) =1=>
<answer part>(*a*, <transition gen action statement>(*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 gen action statement>(*a*, *undefined*)
provided *a.last* $\notin$ <decision> $\wedge$ *t.parentAS0.parentAS0.parentAS0* $\in$ <decision> $\wedge$
t.findContinueLabel $\neq$ *undefined*
=5=> <transition gen action statement>(*a*,
<terminator statement>(*undefined*,<join>(*findContinueLabel*(*t*))))

If a <decision> is not terminating, it is derived syntax for a <decision> where all not terminating <answer part>s and the <else part> (if not terminating) have inserted at the end of their <transition> a <join> to the first <action statement> following the decision or (if the decision is the last <action statement> in a <transition string>) to the following <terminator statement>.

< *d* =<decision>(*, *), <action statement>(*undefined*, *a*) > **provided** $\neg$ *TerminatingDecision*(*d*)
=5=> < *d*, <action statement>(*newName*(*undefined*), *a*) >

<transition gen action statement>(*str*, <terminator statement>(*undefined*, *t*))
provided *str.last* ∈ <decision> ∧ ¬ *str.last.TerminatingDecision*
=5=> <transition gen action statement>(*str*, <terminator statement>(*newName*(*undefined*), *t*))

The rules above insert a new label after a non-terminating decision.

let *nn* = *newName* **in**
d=<decision>(**any**, *exc*, <decision body>(*ans*, *undefined*))
=8=>
<decision>(<expression gen primary>(*undefined*, <any expression>(<identifier>(*empty*, *nn*))), *exc*,
<decision body>(
< <answer part>(<expression gen primary>(*undefined*, *idx*), *ans*[*idx*].s-<transition>)
| *idx* **in** 1 .. *ans.length* >,
undefined))
and
let *parent* = $d.surroundingScopeUnit_0$ **in**
entities=parent.getEntities
=>
entities ⌢ < <syntype definition>(*empty*,
<syntype definition gen 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 <decision body> is followed by N <answer part>s, any in <decision> is then a 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>s are shorthands for writing the literals 1 through N as the <constant>s of the <range condition>s in the N <answer>s.

Mapping to Abstract Syntax

| <decision>(*q*, *exc*, <decision body>(*answers*, *elseAnswer*))
=> **mk-***Decision-node*(*Mapping*(*q*), *Mapping*(*exc*), *Mapping*(*answers*).*toSet*, *Mapping*(*elseAnswer*))

| <answer part>(*ans*, *trans*)
=> **mk-***Decision-answer*(*Mapping*(*ans*), *Mapping*(*trans*))

| <else part>(*trans*)
=> **if** *trans=undefined* **then** *undefined* **else mk-***Else-answer*(*Mapping*(*trans*)) **endif**

Auxiliary Functions

The function $rangeConditionList_0$ gets the list of <range condition>s in a <decision>.

$rangeConditionList_0$(*d*:<decision>):<range condition>*=$_{def}$
let *apl* = *d*.**s-**<decision body>.**s-**<answer part>**-seq in**
$apl.answerPartRangeConditionList_0$
endlet

$answerPartRangeConditionList_0$(*apl*:<answer part>*):<range condition>*=$_{def}$
if *apl.head*.**s-**<answer>∈ <range condition> **then**
apl.head.**s-**<answer>⌢*apl.tail*. $answerPartRangeConditionList_0$
else *apl.tail*. $answerPartRangeConditionList_0$

The function *findContinueLabel* computes the continuation label after a decision within a transition string.

findContinueLabel(*x*: *DefinitionAS0*): <name> =$_{def}$
if *x* ∈ <transition gen action statement> ∧ *x*.**s-**<terminator statement> ≠ *undefined* ∧

x.**s-**<terminator statement>.**s-**<label> = *undefined* ∧
x.**s-**<terminator statement>.**s-**<terminator> ∈ <join>
then *x*.**s-**<terminator statement>.**s-**<terminator>.**s-**<name>
else *findContinueLabel*(*x*.*parentAS0*)
endif

TerminatingDecision(*d*: <decision>): *BOOLEAN* =$_{\text{def}}$
(∀ *a* ∈ *d*.**s-**<answer part>: *TerminatingTransition*(*a*.**s-**<transition>)) ∧
(*d*.**s-**<else part> = *undefined* ∨ *TerminatingTransition*(*d*.**s-**<else part>.**s-**<transition>))

A <decision> is a terminating decision, if each <answer part> and <else part> in its <decision body> is a terminating <answer part> or <else part> respectively.

TerminatingTransition(*t*: <transition>): *BOOLEAN* =$_{\text{def}}$
t ∈ <terminator statement> ∨
t.**s-**<terminator statement> ≠ *undefined* ∨
(**let** *d*= *t*.**s-**<action statement>.*last* **in** *d* ∈ <decision> ∧ *TerminatingDecision* (*d*) **endlet**)

An <answer part> or <else part> in a decision is a terminating <answer part> or <else part> respectively if it contains a <transition> where a <terminator statement> is specified, or contains a <transition string> whose last <action statement> contains a terminating decision.

2.8.14 Statement List

Concrete Syntax

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

<statement> =
<empty statement>
| <compound statement>
| <assignment statement>
| <algorithm action statement>
| <call statement>
| <expression statement>
| <if statement>
| <decision statement>
| <loop statement>
| <terminating statement>
| <labelled statement>
| <exception statement>

<terminating statement> =
<return statement>
| <break statement>
| <loop break statement>
| <loop continue statement>
| <raise statement>

<variable definition statement> :: <local variables of sort>+

<local variables of sort> :: <variable<name>+ <sort> [<expression>]

Conditions on Concrete Syntax

(∀*bs*∈ <loop break statement>: ∃*ls*∈ <loop statement>: *isAncestorAS0*(*ls*,*bs*)) ∧
(∀*cs*∈ <loop continue statement>: ∃*ls*∈ <loop statement>: *isAncestorAS0*(*ls*,*cs*))

A <loop break statement> and <loop continue statement> may only occur within a <loop statement>.

∀*ts*∈ <terminating statement>: ∀ *d* ∈ *DefinitionAS0*:
ts **in** *d*.**s-**<statement>**-seq** ⇒ *ts* = *d*.**s-**<statement>**-seq**.*last*

A <terminating statement> may only occur as the last <statement> in <statements>.

Mapping to Abstract Syntax

< <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>(< *v* > ⌢ *rest*, *s*, *expr*) > **provided** *rest* ≠ *empty* =1=>
< <local variables of sort>(< *v* >, *s*, *expr*), <local variables of sort>(*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

| <statement list>(*vars*, *allstats* = *stats* ⌢ < *terminator* >)
=> **let** *nn*=*newName* **in**
mk-*Compound-node*(*Mapping*(*nn*), *Mapping*(*vars*).*toSet*, *undefined*, *empty*(),
if *terminator* ∈ <terminating statement>
then mk-*Transition*(*Mapping*(*stats*), *Mapping*(*terminator*))
else mk-*Transition*(*Mapping*(*allstats*), **mk-***Break-node*(*Mapping*(*nn*)))
endif,
empty)

| <variable definition statement>(< *var* >) => *Mapping*(*var*)

| <local variables of sort>(< *var* >, *s*, *expr*)
=> **mk-***Variable-definition*(*Mapping*(*var*), *Mapping*(*s*), *Mapping*(*expr*))

2.8.14.1 Compound Statement

Abstract Syntax

Compound-node	::	*Connector-name* *Variable-definition*-**set** [*Exception-handler-node*] *Init-graph-node** *Transition* *Step-graph-node**
Init-graph-node	=	*Graph-node*
Step-graph-node	=	*Graph-node*
Continue-node	::	*Connector-name*
Break-node	::	*Connector-name*

Concrete Syntax

<compound statement> :: <statement list>

Transformations

The following statements are handled by the Mapping.

If the <statement list> 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>s are inserted at the front of <statement>s in the order of their occurrence.

The <statement list> is equivalent to the concatenation of the transform of each <variable definition statement> and the transform of each <statement> in <statements> (see 11.14.1 to 11.14.7).

NOTE – The transformed non-empty <statement list> becomes a list of <action statement>s and <terminator statement>s separated by semicolons and ending in a semicolon and therefore can be treated as a <transition>.

If the <statement list> is empty, the result of its transformation is the empty text.

Mapping to Abstract Syntax

| <compound statement>(s) => *Mapping*(s)

2.8.14.2 Transition Actions and Terminators as Statements

Concrete Syntax

<assignment statement> :: <assignment>
<algorithm action statement> =
<output body>
| <create body>
| <set body>
| <reset body>
| <export body>
<return statement> :: <return body>
<raise statement> :: <raise body>
<call statement> :: <procedure call body>

Conditions on Concrete Syntax

∀*rs*∈ <return statement>:
(*parentAS0ofKind*(*rs*, <procedure definitions>)≠*undefined*)∨
(*parentAS0ofKind*(*rs*,<operation definition>)≠*undefined*)

A <return statement> is only allowed within a <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>, a <return statement>, and <raise statement> is obtained by dropping the trailing <end>.

Mapping to Abstract Syntax

| <assignment statement>(*a*) => *Mapping*(*a*)

| <return statement>(*r*) => *Mapping*(*r*)

| <raise statement>(*r*) => *Mapping*(*r*)

| <call statement>(*c*) => *Mapping*(*c*)

2.8.14.3 Expressions as Statements

Concrete Syntax

<expression statement> :: <operator application>

Transformations

let *nn*=*newName*() **in**
<expression statement>(*expr*) =3=>
<compound statement>(<statement list>(
< <variable definition statement>(< <local variables of sort>(< *nn* >, *expr*.*staticSort*$_0$, *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>.

2.8.14.4 If Statements

Concrete Syntax

<if statement> :: <Boolean<expression> <consequence statement> [<alternative statement>]

<consequence statement> = <statement>

<alternative statement> = <statement>

Transformations

<if statement>(*expr*, *cons*, *alt*) =3=>
<decision statement>(*expr*,
<decision statement body>(<
<algorithm answer part>("True", *cons*),
<algorithm answer part>("False", **if** *alt* = *undefined* **then** <empty statement> **else** *alt* **endif**)
>, *undefined*))

The <if statement> is equivalent to the following <action statement> involving a <decision>:

decision <Boolean expression> ;
(true) : task { <consequence statement>-transform };
(false) : task { <alternative statement>-transform };
enddecision ;

The transform of <alternative statement> is only inserted if <alternative statement> was present.

2.8.14.5 Decision Statements

Concrete Syntax

<decision statement> :: <expression> <decision statement body>
<decision statement body> :: <algorithm answer part>+ [<algorithm else part>]
<algorithm answer part> :: <range condition> <statement>
<algorithm else part> :: <alternative statement>

Transformations

The following statements are handled by the Mapping.

An <algorithm answer part> is transformed into the following <answer part>.

(<range condition>) : task { <statement>-transform };

A <decision body> is then formed by taking the transform of each <algorithm answer part> in order and appending the following <else part>.

else : task { <alternative statement>-transform };

where the transformation of <alternative statement> is only inserted if <alternative statement> is present. The resulting <decision body> is referred to as Body. The <decision statement> is equivalent to the follow-ing <action statement>:

decision (<expression>) ;

Body

enddecision ;

Mapping to Abstract Syntax

```
| <decision statement>(q, <decision statement body>(answers, elseAnswer))
    => let nn=newName() in
        mk-Compound-node(Mapping(nn), ∅, empty(),
            mk-Transition(
                mk-Decision-node(Mapping(q), undefined,
                    { mk-Decision-answer(Mapping(a.s-<range condition>),
                        if a.s-<statement> ∈ <terminating statement>
                            then mk-Transition(empty(), Mapping(a.s-<statement>))
                        elsif a.s-<statement> ∈ <empty statement>
                            then mk-Transition(empty(), mk-Break-node(Mapping(nn)))
                        else mk-Transition(< Mapping(a.s-<statement>) >, mk-Break-node(Mapping(nn)))
                        endif
                    ) },
                    Mapping(elsePart))
            )
        )
        endlet

| <algorithm else part>(trans) => mk-Else-answer(Mapping(trans))
```

2.8.14.6 Loop Statements

Concrete Syntax

```
<loop statement> ::
        <loop clause>* <loop body statement> [<finalization statement>]

<loop body statement> = <statement>

<finalization statement> = <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 gen identifier>

<loop variable indication gen identifier> :: <variable<identifier> [<expression>]

<loop variable definition> :: <variable<name> <sort> <expression>
```

<loop break statement> :: ()

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

Conditions on Concrete Syntax

∀*ls*∈ <loop step>:
(**let** *pd* = *ls*.**s-**<procedure call body>.*calledProcedure*$_0$ **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>(*, *, *) **provided** l.*parentAS0* ∉ <labelled statement> =3=>
<labelled statement>(<label>(*newName*), *l*)

Generate a name for every unlabelled <loop statement>.

l=<loop break statement> =3=>
<break statement>(*parentAS0ofKind*(*l*, <labelled statement>).**s-**<label>.**s-**<name>)

Every occurrence of a <loop break statement> inside a <loop clause> or the <loop body statement> or a <finalization statement> of another <loop statement> contained within this <loop statement>, all not occurring within another inner <loop statement>, is replaced by

break Label ;

The following condition is satisfied by the definition of the function bigAnd.

If a <Boolean expression> is absent in a <loop clause>, the predefined Boolean value true is inserted as the <Boolean expression>.

The following transformation is already handled by the Mapping.

Then the <loop statement> is replaced by the so modified <loop statement> followed by a <labelled statement> with <connector name> Break.

Mapping to Abstract Syntax

| <loop statement>(*cl*, *body*, *final*)
=> **let** *nn*=*newName*() **in**
mk-*Compound-node*(*Mapping*(*nn*),
{ **mk-***Variable-definition*(*Mapping*(*c*.**s-**<loop variable indication>.**s-**<name>),
Mapping(*c*.**s-**<loop variable indication>.**s-**<sort>), *undefined*) |
c ∈ *cl*.*toSet*: *c*.**s-**<loop variable indication> ∈ <loop variable definition> },
< *Mapping*(*c*.**s-**<loop variable indication>) | *c* **in** *cl*: *c*.**s-**<loop variable indication> ≠ *undefined* >,
mk-*Decision-node*(*Mapping*(*bigAnd*(< *c*.**s-**<expression>) | *c* **in** *cl*: *c*.**s-**<expression> ≠ *undefined* >)),
undefined, { **mk-***Decision-answer*("True",
if *body* ∈ <terminating statement>
then mk-*Transition*(*empty*(), *Mapping*(*body*))
elsif *body* ∈ <empty statement>
then mk-*Transition*(*empty*(), **mk-***Continue-node*(*Mapping*(*nn*)))
else mk-*Transition*(< *Mapping*(*body*) >, **mk-***Continue-node*(*Mapping*(*nn*)))
endif,
mk-*Decision-answer*("False",
mk-*Transition*(< *Mapping*(*final*) >, **mk-***Break-node*(*Mapping*(*nn*))))
}, *undefined*),
Mapping(*cl*))

| <loop variable definition>(*n*, *, *expr*)
=> **mk-***Graph-node*(**mk-***Assignment*(*Mapping*(*n*), *Mapping*(*expr*)), *undefined*)

| <loop variable indication gen identifier>(*id*, *expr*)

=> **mk-***Graph-node*(**mk-***Assignment*(*Mapping*(*id*), *Mapping*(*expr*)), *undefined*)

| <loop clause>(*ind*, *, *expr*)
=> **if** *c* ∈ <expression> **then**
mk-*Graph-node*(**mk-***Assignment*(
if *ind* ∈ <loop variable definition>
then *Mapping*(*ind*.**s-**<name>)
else *Mapping*(*ind*.**s-**<identifier>) **endif**,
Mapping(*expr*)), *undefined*)
else *Mapping*(*c*) **endif**

Auxiliary Functions

bigAnd(*seq*: <expression>*): <expression> =$_{def}$
if *seq=empty* **then** "True"
elseif *seq*. *length*=1 **then** *seq*. *head*
else <operator application>(*seq*. *head*, *bigAnd*(*seq.tail*))
endif

2.8.14.7 Break and Labelled Statements

Concrete Syntax

<break statement> :: <connector<name>

<labelled statement> :: <label> <statement>

Conditions on Concrete Syntax

∀*bs*∈ <break statement>: ∃*ls*∈ <labelled statement>:
isAncestorAS0(*ls*.**s-**<statement>, *bs*) ∧ (*bs*.**s-**<name> = *ls*.**s-**<label>.**s-**<name>)

A <break statement> must be contained in a statement that has been labelled with the given <connector name>.

Mapping to Abstract Syntax

| <break statement>(*name*) => **mk-***Break-node*(*Mapping*(*name*))

| <labelled statement>(*name*, *body*)
=> **mk-***Compound-node*(*Mapping*(*name*), ∅, *empty*,
if *body* ∈ <terminating statement>
then mk-*Transition*(*empty*(), *Mapping*(*body*))
elsif *body* ∈ <empty statement>
then mk-*Transition*(*empty*(), **mk-***Continue-node*(*Mapping*(*nn*)))
else mk-*Transition*(< *Mapping*(*body*) >, **mk-***Continue-node*(*Mapping*(*nn*)))
endif,
empty())

2.8.14.8 Empty Statement

Concrete Syntax

<empty statement> :: ()

Transformations

< <empty statement> > =1=> *empty*

The transform of the <empty statement> is the empty text.

2.8.14.9 Exception Statement

Concrete Syntax

<exception statement> :: <try statement> <handle statement>+

<try statement> =<statement>

<handle statement> :: <exception stimulus list> <statement>

Conditions on Concrete Syntax

$\forall ts \in$ <try statement>: $ts \notin$ <break statement>

The <try statement> must not be a <break statement>.

Transformations

The following statement is handled by the mapping below.

If the <try statement> was not a <compound statement>, the <try statement> is first transformed into a <compound statement>:

{ <try statement> }

Then the (transformed) <try statement> and all <handle statement>s are transformed. For each <handle statement>, the following <handle> is constructed

handle <exception stimulus list> ;

task { <handle statement>-transform };

The constructed <handle>s are collected into a list referred to as HandleParts. An exception handler is constructed having an anonymous name. The name of the exception handler is referred to as handler.

exceptionhandler handler;

HandleParts

endexceptionhandler;

The transform of the <exception statement> is the transform of the <try statement>.

Mapping

| *e*=<exception statement>(*try*, *handles*) =>
let *nn*=*newName*() **in**
mk-*Graph-node*(
mk-*Compound-node*(*nn*, ∅,
mk-*Exception-handler-node*(*nn*, *undefined*, *Mapping*(*handles*), *undefined*)
empty, *Mapping*(*try*), *empty*),
mk-*On-exception*(*nn*))

| <handle statement>(< <exception stimulus>(*id*, *vars*) >, *stmt*) =>
mk-*Handle-node*(*Mapping*(*id*), *Mapping*(*vars*), *undefined*, *Mapping*(*stmt*))

2.8.15 Timer

Abstract Syntax

Timer-definition	::	*Timer-name Sort-reference-identifier**
Set-node	::	*Time-expression Timer-identifier Expression**
Time-expression	=	*Expression*
Reset-node	::	*Timer-identifier Expression**

Conditions on Abstract Syntax

$\forall n \in$ *Set-node* $\cup$ *Reset-node*: $\forall d \in$ *Timer-definition*:
(d=*getEntityDefinition*$_1$(n.**s**-*Timer-identifier*,**timer**)) $\Rightarrow$
isActualAndFormalParameterMatched$_1$(n.**s**-*Expression*-**seq**, d.*formalParameterSortList*$_1$)

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>* [<timer default initialization>]

<timer default initialization> :: <Duration<constant expression>

<set> :: <set body> [<on exception>]

<set body> :: <set clause>+

<set clause> :: [<Time<expression>] <timer<identifier> <expression>*

<reset> :: <reset body> [<on exception>]

<reset body> :: <reset clause>+

<reset clause> :: <timer<identifier> <expression>*

Transformations

<set clause>(*undefined*, *id*, *exprList*)
=8=> <set clause>(<operator application>("+", **now**, *id.refersto*$_0$.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 > $^\frown$ *rest*) > **provided** *rest* $\neq$ *empty* =1=>
< <set>(< s >), <set>(*rest*) >

< <reset>(< r > $^\frown$ *rest*) > **provided** *rest* $\neq$ *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 shorthands in the contained expressions are expanded.

< <timer definition>(< t > $^\frown$ *rest*) > **provided** *rest* $\neq$ *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* >, *exc*) => **mk-***Graph-node*(*Mapping*(*clause*), *Mapping*(*exc*))

| <set clause>(*expr*, *id*, *params*) => **mk-***Set-node*(*Mapping*(*expr*), *Mapping*(*id*), *Mapping*(*params*))

| <reset>(< *clause* >, *exc*) => **mk-***Graph-node*(*Mapping*(*clause*), *Mapping*(*exc*))

| <reset clause>(*id*, *params*) => **mk-***Reset-node*(*Mapping*(*id*), *Mapping*(*params*))

2.8.16 Exception

2.8.16.1 Exception Definition

Abstract Syntax

Exception-definition :: *Exception-name*
*Sort-reference-identifier***

Concrete Syntax

<exception definition> :: <exception definition item>+

<exception definition item> ::
<exception><name> <sort>*

Transformations

< <exception definition>(< *t* > ⌢ *rest*) > **provided** *rest* ≠ *empty* =1=>
< <exception definition>(< *s* >), <exception definition>(*rest*) >

An <exception definition> may contain several <exception definition item>s. This is derived syntax for specifying a sequence of <exception definitions>s, one for each <exception definition item>. This is an auxiliary transformation.

Mapping to Abstract Syntax

| <exception definition>(< *item* >) => *Mapping*(*item*)

| <exception definition item>(*name*, *sortList*) => **mk-***Exception-definition*(*Mapping*(*name*), *Mapping*(*sortList*))

2.8.16.2 Exception Handler

Abstract Syntax

Exception-handler-node :: *Exception-handler-name*
[*On-exception*]
*Handle-node***-set**
[*Else-handle-node*]

Conditions on Abstract Syntax

∀*n*, *n′* ∈ *Exception-handler-node*:
(*parentAS1ofKind*(*n*, *State-transition-graph*∪*Procedure-graph*)=
parentAS1ofKind(*n′*, *State-transition-graph*∪*Procedure-graph*))∧ *n* ≠ *n′* ⇒
n.**s-***Exception-handler-name* ≠ *n′*.**s-***Exception-handler-name*

The *Exception-handler-node*s within a given *State-transition-graph* or *Procedure-graph* must all have different *Exception-handler-name*.

Concrete Syntax

<exception handler> ::
<exception handler list> [<on exception>] <handle>*

<exception handler list> =

<exception handler><name>+ | <asterisk exception handler list>

<asterisk exception handler list> ::
<exception handler><name>*

Conditions on Concrete Syntax

∀*eh*∈ <exception handler>: *eh*.**s-**<exception handler list>∈ <asterisk exception handler list>⇒
let *aehl* = *eh*.**s-**<exception handler list>.**s-**<name>**-seq in**
(∀*i*,*j*∈ 1..*aehl.length*: *i*≠*j*⇒*aehl*[*i*]≠ *aehl*[*j*])∧
(∀*n*∈ <name>: *n* **in** *aehl*⇒ *n*∈ *eh.exceptionHandlerNameSet*$_0$)

The <exception handler name>s in an <asterisk exception handler list> must be distinct and must be contained in other <exception handler list>s in the enclosing body or in the body of a supertype.

∀*eh1*,*eh2*∈ <exception handler>: *eh1.surroundingScopeUnit*$_0$=*eh2.surroundingScopeUnit*$_0$⇒
(*eh1.exceptionHandlerNameSet*$_0$∩*eh2.exceptionHandlerNameSet*$_0$=∅∨
eh1.**s-**<on exception>.**s-**<name>≠*eh2*.**s-**<on exception>.**s-**<name>)

An <exception handler> has at most one <exception handler> associated.

Transformations

< <exception handler>(< *t* > ⌢ *rest*, *exc*, *h*, *undefined*) > **provided** *rest* ≠ *empty* =1=>
< <exception handler>(< *t* >, *exc*, *h*, *undefined*), <exception handler>(*rest*, *exc*, *h*, *undefined*) >

When the <exception handler list> of an <exception handler> contains more than one <exception handler name>, a copy of the <exception handler> is created for each such <exception handler name>. Then the <exception handler> is replaced by these copies.

e=<exception handler>(<asterisk exception handler list>(*except*), *exc*, *h*, *undefined*) > =13=>
<exception handler>(< *ex* **in** *e.surroundingScopeUnit*$_0$.*exceptionHandlerSet*$_0$: (*ex* ∉ *except.toSet*) >,
exc, *h*, *undefined*)

An <exception handler> with an <asterisk exception handler list> is transformed to a list of <exception handler>s, one for each <exception handler name> of the body in question, except for those <exception handler name>s contained in the <asterisk exception handler list>.

< <exception handler>(< *s* >, *exc1*, *handles1*, *) > ⌢ *rest* ⌢ < <exception handler>(< *s* >, *exc2*, *handles2*, *) >
=13=> *rest* ⌢ < <exception handler>(< *s* >, **if** *exc1* ≠ *undefined* **then** *exc1* **else** *exc2* **endif**,
handles1 ⌢ *handles2*, *undefined*) >

When several <exception handler>s contain the same <exception handler name>, these <exception handler>s are concatenated into one <exception handler> having that <exception handler name>.

Mapping to Abstract Syntax

| <exception handler>(< *name* >, *exc*, *handles*, *)
=> **mk-***Exception-handler-node*(*Mapping*(*name*), *Mapping*(*exc*),
{ *h* ∈ *Mapping*(*handles*).*toSet*: *h* ∈ *Handle-node* },
head(< *h* **in** *Mapping*(*handles*): (*h* ∈ *Else-handle-node*) >))

Auxiliary Functions

The function *exceptionHandlerSet*$_0$ gets the set of <exception handler>s defined in the given definition.

exceptionHandlerSet$_0$(*d*: <agent type definition> ∪<agent definition> ∪<procedure definitions>∪
<composite state type definition>∪<composite state> ∪
<operation definition>): <exception handler>**-set** =$_{def}$
d.localExceptionHandlerSet$_0$∪*d.inheritedExceptionHandlerSet*$_0$

localExceptionHandlerSet$_0$(*d*: <agent type definition> ∪<agent definition> ∪<procedure definitions>∪
<composite state type definition>∪<composite state> ∪
<operation definition>): <exception handler>-**set** =$_{def}$
{*eh* ∈ <exception handler>: *eh.surroundingScopeUnit*$_0$ = *d*}

inheritedExceptionHandlerSet$_0$(*d*: <agent type definition> ∪<agent definition> ∪
<procedure definitions>∪<composite state type definition>∪<composite state> ∪
<operation definition>): <exception handler>-**set** =$_{def}$
if (*d.specialization*$_0$=*undefined*) **then** ∅
else *d.specialization*$_0$.**s-**<type expression>.*baseType*$_0$.*exceptionHandlerSet*$_0$
endif

The functions *associatedExceptionHandler0* gets the <exception handler>s associated with a exception handler name.

associatedExceptionHandler$_0$(*hn*: <name>): <exception handler> =$_{def}$
take({*eh*∈ <exception handler>: *hn*∈ *eh.exceptionHandlerNameSet*$_0$})

2.8.16.3 On-Exception

Abstract Syntax

On-exception :: *Exception-handler-name*

Conditions on Abstract Syntax

∀*oe* ∈ *On-exception*: ∃*hn* ∈ *Exception-handler-node*:
(*oe*.**s-***Exception-handler-name* = *hn*.**s-***Exception-handler-name*) ∧
(*parentAS1ofKind*(*oe*,*State-transition-graph* ∪ *Procedure-graph*)=
parentAS1ofKind(*hn*,*State-transition-graph* ∪ *Procedure-graph*))

The *Exception-handler-name* specified in *On-exception* must be the name of an *Exception-handler-node* within the same *State-transition-graph* or *Procedure-graph*.

Concrete Syntax

<on exception> :: <exception handler><name>

Transformations

The following statement is formalised in the dynamic semantics.

In a specialization, the association with the exception handler is considered as a part of the graph or the transition. If a virtual transition is redefined, the new transition replaces an <on exception> of the original transition. If a graph or a state is inherited in a specialization, any associated exception handler is inherited as well.

Mapping to Abstract Syntax

| <on exception>(*name*) => **mk-***On-exception*(*Mapping*(*name*))

2.8.16.4 Handle

Abstract Syntax

Handle-node :: *Exception-identifier*
[*Variable-identifier*]*
[*On-exception*]
Transition

Else-handle-node :: [*On-exception*] *Transition*

Conditions on Abstract Syntax

∀*hn*,*hn'* ∈ *Handle-node*:
hn. *parentAS1*= *hn'*. *parentAS1*∧ *hn*.**s-***Exception-identifier* = *hn'*.**s-***Exception-identifier*⇒ *hn* = *hn'*

The *Exception-identifier*s in the *Handle-node*-**set** must be distinct.

∀*hn* ∈ *Handle-node*:
(**let** *d* = *getEntityDefinition*$_1$(*hn*.**s-***Exception-identifier*, **exception**) **in**
isActualAndFormalParameterMatched$_1$(*hn*.**s-***Variable-identifier*-**seq**, *d*.*formalParameterSortList*$_1$)
endlet)

The length of the list of optional *Variable-identifier*s in *Handle-node* must be the same as the number of *Sort-reference-identifier*s in the *Exception-definition* denoted by the *Exception-identifier* and the sorts of the variables (if any) must correspond by position to the sorts of the data items that can be carried by the exception.

Concrete Syntax

<handle> ::
[<virtuality>] <exception stimulus list> [<on exception>] <transition>

<exception stimulus list> =
<exception stimulus>+ | <asterisk exception stimulus list>

<exception stimulus> ::
<exception><identifier> [<variable>]*

<asterisk exception stimulus list>:: ()

Transformations

< <handle>(*virt*, < *s* > ⌢ *rest*, *exc*, *trans*) > **provided** *rest* ≠ *empty* =1=>
< <handle>(*virt*, < *s* >, *exc*, *trans*), <handle>(*virt*, *rest*, *exc*, *trans*) >

When the <exception stimulus list> of a certain <handle> contains more than one <exception stimulus>, a copy of the <handle> is created for each <exception stimulus>. Then the <handle> is replaced by these copies.

<handle>(*virt*, < <exception stimulus>(*id*, *vars*) >, *exc*, <transition gen action statement>(*actions*, *term*))
provided { *v* ∈ *vars*.*toSet*: *v* ∈ (<indexed variable> ∪ <field variable>) } ≠ ∅
=8=>
(**let** *newvars* =
< (**if** *v* ∈ (<indexed variable> ∪ <field variable>) **then** *newName*(*v*) **else** *v* **endif**) | *v* **in** *vars* > **in**
let *newtrans* = <transition gen action statement>(
< <action statement>(*undefined*, <task>(<assignment>(*v*, *newName*(*v*))), *undefined*)
| *v* **in** *vars*: (*v* ∈ <indexed variable> ∪ <field variable>)>
⌢ *actions*, *term*) **in**
<handle>(*virt*, < <exception stimulus>(*item*, *newvars*) >, *exc*, *newtrans*)
endlet)

When one or more of the <variable>s of a certain <exception stimulus> are <indexed variable>s or <field variable>s, all the <variable>s are replaced by unique, new, implicitly declared <variable identifier>s. Immediately before the <transition> of the <handle>, a <task> is inserted which in its <textual task body> contains an <assignment> for each of the <variable>s, assigning the result of the corresponding new variable to the <variable>. The results are assigned in the order from left to right of the list of <variable>s. This <task> becomes the first <action statement> in the <transition>.

Mapping to Abstract Syntax

| <handle>(*, < <exception stimulus>(*id*, *vars*) >, *exc*, *trans*)
=> **mk-***Handle-node*(*Mapping*(*id*), *Mapping*(*vars*), *Mapping*(*exc*), *Mapping*(*trans*))

| <handle>(*, <asterisk>, *exc*, *trans*)
=> **mk-***Else-handle-node*(*Mapping*(*exc*), *Mapping*(*trans*))

2.9 Data

2.9.1 Data Definitions

Concrete Syntax

<data definition> =
    <data type definition> | <interface definition> | <syntype definition> | <synonym definition>

<sort> = <basic sort> | <anchored sort> | <expanded sort> | <reference sort> | <pid sort>

<basic sort> = <sort<identifier> | <syntype>

<anchored sort> :: [<basic sort>]

<expanded sort> :: <basic sort>

<reference sort> :: <basic sort>

<pid sort> = <sort<identifier>

Transformations

<expanded sort>(*sort*) **provided** ¬ *sort.isObjectSort*$_0$
    =8=> *sort*

An <expanded sort> with a <basic sort> that represents a value sort is replaced by the <basic sort>.

<expanded sort>(*sort*) **provided** *sort.isObjectSort*$_0$

    =8=> *sort* A <reference sort> with a <basic sort> that represents an object sort is replaced by the <basic sort>.

NOTE – As a consequence, the keyword value has no effect if the sort has been defined as a set of values, and the keyword object has no effect if the sort has been defined as a set of objects.

An <anchored sort> without a <basic sort> is a shorthand for specifying a <basic sort> referencing the name of the data type definition or syntype definition in the context of which the <anchored sort> occurs.

Mapping to Abstract Syntax

| <expanded sort>(*ident*) => **mk-***Expanded-sort*(*Mapping*(*ident*))

| <reference sort>(*ident*) => **mk-***Reference-sort*(*Mapping*(*ident*))

2.9.2 Data Type Definition

Abstract Syntax

Data-type-definition	=	*Value-data-type-definition*
	|	*Object-data-type-definition*
	|	*Interface-definition*
Value-data-type-definition	::	*Sort*
		Data-type-identifier
		Literal-signature-**set**
		Static-operation-signature-**set**
		Dynamic-operation-signature-**set**
		Data-type-definition-**set**
		Syntype-definition-**set**
		Exception-definition-**set**
Object-data-type-definition	::	*Sort*
		Data-type-identifier

Literal-signature-**set**
Static-operation-signature-**set**
Dynamic-operation-signature-**set**
Data-type-definition-**set**
Syntype-definition-**set**
Exception-definition-**set**

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 kind> <data type<name> <formal context parameter>* [<virtuality constraint>]

<data type kind> :: **value** | **object**

<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> | <exception definition>

<operations> :: <operation signatures> <textual operation reference>* <operation definitions>*

Conditions on Concrete Syntax

$\forall$*dtd*$\in$ <data type definition>:$\forall$*fcp*$\in$ <formal context parameter>:
fcp$\in$ *dtd.localFormalContextParameterList*$_0$*.toSet* $\Rightarrow$
fcp$\in$ <sort context parameter> $\cup$<synonym context parameter>

A <formal context parameter> of <formal context parameters> must be either a <sort context parameter> or a <synonym context parameter>.

$\forall$*anchSort*$\in$ <anchored sort>:
parentAS0ofKind(*anchSort*, <data type definition>) $\neq$ *undefined* $\wedge$ (*anchSort*.**s-**<basic sort> $\neq$ *undefined*$\Rightarrow$
parentAS0ofKind(*anchSort*, <data type definition>) =
getEntityDefinition$_0$(*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>.

$\forall$*sid*$\in$ <pid sort>:*isPidSort*$_0$(*sid*)

The <sort identifier> in a <pid sort> must reference a pid sort.

Transformations

< *s* = <system type definition>(*, <system type heading>(*name*, *heading*)) >
=9=>
< *s* > $^\frown$
< <interface definition>(*undefined*,<interface heading>(*name*, <>, *undefined*),
<interface specialization>(
< <type expression>(<identifier>(<path item>(**system type**, *name*), *name*)) >), <>, *undefined*) >

< *s* = <block type definition>(*, <block type heading>(*name*, *heading*)) >
=9=>
< *s* > $^\frown$
< <interface definition>(*undefined*,<interface heading>(*name*, <>, *undefined*),
<interface specialization>(
< <type expression>(<identifier>(<path item>(**block type**, *name*), *name*)) >), <>, *undefined*) >

< *s* = <process type definition>(*, <process type heading>(*name*, *heading*)) >
=9=>
< *s* > ⌢
< <interface definition>(*undefined*,<interface heading>(*name*, <>, *undefined*),
<interface specialization>(
< <type expression>(<identifier>(<path item>(**process type**, *name*), *name*)) >), <>, *undefined*) >

< *s* = <textual typebased system definition>(
<typebased system heading>(name, <type expression>(base, *))) > =9=>
< *s* > ⌢
< <interface definition>(*undefined*,<interface heading>(*name*, <>, *undefined*),
<interface specialization>(< <type expression>(*base*) >), <>, *undefined*) >

< *s* = <textual typebased block definition>(
<typebased block heading>(name, <type expression>(base, *))) > =9=>
< *s* > ⌢
< <interface definition>(*undefined*,<interface heading>(*name*, <>, *undefined*),
<interface specialization>(< <type expression>(*base*) >), <>, *undefined*) >

< *s* = <textual typebased process definition>(
<typebased process heading>(name, <type expression>(base, *))) > =9=>
< *s* > ⌢
< <interface definition>(*undefined*,<interface heading>(*name*, <>, *undefined*),
<interface specialization>(< <type expression>(*base*) >), <>, *undefined*) >

a = <agent structure>(*iset*, *entities*, *interaction*)=9=>
<agent structure>(*iset*, *entities* ⌢
< <interface definition>(*undefined*,<interface heading>(*name*, <>, *undefined*),
<interface specialization>(*a.gateDefinitionSet*$_0$*.impliedBases*, <>,
a.gateDefinitionSet$_0$*.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 1 – 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 2 – 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>(**value**, *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* ∈ *Dynamic-operation-signature* },
{ *e* ∈ *Mapping*(*entities*).*toSet*: *e* ∈ *Data-type-definition* },
{ *e* ∈ *Mapping*(*entities*).*toSet*: *e* ∈ *Syntype-definition* },
{ *e* ∈ *Mapping*(*entities*).*toSet*: *e* ∈ *Exception-definition* })

| <data type definition>(*, *, <data type heading>(**object**, *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* ∈ *Dynamic-operation-signature* }
{ *e* ∈ *Mapping*(*entities*).*toSet*: *e* ∈ *Data-type-definition* },
{ *e* ∈ *Mapping*(*entities*).*toSet*: *e* ∈ *Syntype-definition* },
{ *e* ∈ *Mapping*(*entities*).*toSet*: *e* ∈ *Exception-definition* })

| <data type definition body>(*entities*, *ctor*, *operations*, *) => *Mapping*(*entities*) ⌢ *Mapping*(*ctor*)

| <operations>(*, *bodies*) => *Mapping*(*bodies*)

Auxiliary Functions

The function *localDataTypeDefinitionSet*$_0$ gets the <data type definition>s defined locally in a scope unit.

localDataTypeDefinitionSet$_0$(*su*: *SCOPEUNIT*$_0$): <data type definition>-**set**=$_{def}$
{*d*∈ <data type definition>: *d.surroundingScopeUnit*$_0$ = *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>): *SIGNAL*$_0$=$_{def}$
case *g* **of**
| < <textual gate definition>(*, *c1*, *c2*) > ⌢ *tail* =>
gatesignalset0(*c1*) ∪ *gatesignalset0*(*c2*) ∪ *gatesignalset0* (*tail*)
| < <textual interface gate definition> (*, *ident*) > ⌢ *tail* =>
let *fd* = *getEntityDefinition*$_0$(*ident*, **interface**) **in**
fd.usedSignalSet$_0$∪{*sd.identifier*$_0$: *sd*∈ *fd.definedSignalSet*$_0$} ∪ *tail.gatesignalset0*
| <gate constraint>(**in**, *, *signals*) => *signals.signalSet*$_0$
otherwise
∅
endcase

2.9.3 Interface Type

Abstract Syntax

Interface-definition	::	*Sort* *Data-type-identifier** *Signal-definition*-**set** *Exception-definition*-**set**
Sort	=	*Name*

Concrete Syntax

<interface definition> ::
<package use clause>* [<virtuality>] <interface heading> [<interface specialization>]
<entity in interface>* <interface use list>

<interface heading> ::
<interface><name> <formal context parameter>* [<virtuality constraint>]

<entity in interface> =
<signal definition>
| <interface variable definition>
| <interface procedure definition>
| <exception definition>

<interface use list> :: <signal list item>*

<interface variable definition> :: <remote variable><name>+ <sort>

<interface procedure definition> :: <remote procedure><name> <procedure signature>

Conditions on Concrete Syntax

$\forall fd \in$ <interface definition>: $isRestrictedByInterface_0(fd.$**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.

$\forall ind \in$ <interface definition>:$\forall fcp \in$ <formal context parameter>:
$fcp \in ind.localFormalContextParameterList_0 \Rightarrow$
$fcp \in$ <signal context parameter> $\cup$<remote procedure context parameter> $\cup$
<remote variable context parameter> $\cup$ <sort context parameter>$\cup$<exception context parameter>

The <formal context parameters> shall only contain <signal context parameter>, <remote procedure context parameter>, <remote variable context parameter>, <sort context parameter> or <exception context parameter>.

Mapping to Abstract Syntax

| <interface definition>(*, *, <interface heading>(*name*, *, *), *spec*, *entities*, *)
=> **mk-***Interface-definition*(*Mapping*(*name*), *Mapping*(*spec*)
{ $e \in$ *Mapping*(*entities*).*toSet*: ($e \in$ *Signal-definition*)},
{ $e \in$ *Mapping*(*entities*).*toSet*: ($e \in$ *Exception-definition*)})

Auxiliary Functions

The function $localInterfaceDefinitionSet_0$ gets the local defined interface definition list of a scope unit.

$localInterfaceDefinitionSet_0$(*su*: $SCOPEUNIT_0$): <interface definition>**-set**$=_{def}$
{$d \in$ <interface definition>:$d.surroundingScopeUnit_0 = su$}

The function $isRestrictedByInterface_0$ 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.

$isRestrictedByInterface_0$(*sl*: <signal list item>*): $BOOLEAN =_{def}$
if *sl* = *empty* **then** *True*
else
case *sl.head*.**s-implicit of**
| {**signal**, **remote procedure**, **remote variable**}=>$isRestrictedByInterface_0$(*sl.tail*)
| {**signallist**}=>
let *sl'* = $getEntityDefinition_0$(*sl.head*.**s-**<identifier>, **signallist**).**s-**<signal list item>**-seq in**
$isRestrictedByInterface_0$(*sl'*)$\wedge$$isRestrictedByInterface_0$(*sl.tail*)

otherwise=>*False*
endcase
endif

2.9.4 Specialization of Data Types

Concrete Syntax

<data type specialization> :: <data type><type expression> <renaming>

<renaming> :: <rename pair>*

<rename pair> =
<rename pair gen operation name> | <rename pair gen literal name>

<rename pair gen operation name> :: <operation name> <base type><operation name>

<rename pair gen 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>:
isSubtype$_0$(*dataDef*, *superTypeDef*) ⇒
superTypeDef.**s-**<data type definition body>.**s-**<data type constructor>= *undefined* ∨
isSameConstructorKind$_0$(*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>.*baseType*$_0$ **in**
let *onl* = <*rp*.**s2-**<operation name> |
rp **in** *sp*. **s-**<renaming>.**s-**<rename pair>**-seq**: *rp*∈ <rename pair gen operation name>> **in**
∀*on*∈ <operation name> : *on* **in** *onl*⇒
(∃*os*∈ <operation signature> : *os*.*surroundingScopeUnit*$_0$ = *bt* ∧ *on* = *os*.*name*$_0$)
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>.

Transformations

<data type definition>(*use*, *preamble*, *heading*, *spec*, *undefined*)
=1=>
<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*))
=1=>

<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 8.2.14 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 8.3.1, unless the operator or method has been declared as private (see 12.1.9.3) 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 the subsort. For inherited virtual methods, <argument virtuality>is added to an <argument> containing an <anchored sort>, if it is not already present.

NOTE – According to the model for specialization in 8.2.14, an operator is only inherited if its signature contained at least one <anchored sort> or renaming had taken place.

Mapping to Abstract Syntax

| <data type specialization>(*base*, *) => *Mapping*(*base*)

| <interface specialization>(*bases*, *) => *Mapping*(*bases*)

Auxiliary Functions

The function *isRenamedBy*$_0$ determine if a <literal signature> or an <operation signature> is renamed by a <specialization>.

isRenamedBySpec$_0$(*sn*:<literal signature>∪<operation signature> , *spec*:<specialization>): *BOOLEAN*=$_{\text{def}}$
(∃*rp*∈ <rename pair>:
(*rp*. *parentAS0.parentAS0 = spec*)∧
(*rp*.**s2**-<literal name> = *sn.name*$_0$ ∨ *rp*.**s2**-<operation name> = *sn.name*$_0$))

The function *isSameConstructorKind*$_0$ is used to determine if two data type constructor is of the same kind.

isSameConstructorKind$_0$(*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 *isSortCompatible*$_0$ is used to determine if the first sort is sort compatible to the second one.

isSortCompatible$_0$(*sort1*:<sort>, *sort2*:<sort>): *BOOLEAN* =$_{\text{def}}$
isSameSort$_0$(*sort1*, *sort2*) ∨
isDirectlySortCompatible$_0$(*sort1*, *sort2*) ∨
((*isObjectSort*$_0$(*sort2*)∨ *isPidSort*$_0$(*sort2*)) ∧

∃*sort3*∈ <sort>: *isSortCompatible*$_0$(*sort1*, *sort3*) ∧ *isSortCompatible*$_0$(*sort3*, *sort2*))

The function *isSameSort*$_0$ is used to determine if the given two sorts are the same.

isSameSort$_0$(*sort1*: <sort>, *sort2*: <sort>): *BOOLEAN* =$_{def}$
sort1 = *sort2* ∨
(*sort1*∈ <basic sort>∧ *sort2*∈ <basic sort>∧
getEntityDefinition$_0$(*sort1*,**sort**).*derivedDateType*$_0$=*getEntityDefinition*$_0$(*sort2*, **sort**).*derivedDateType*$_0$)∨
(*sort1*∈ <anchored sort> ∧ *sort2*∈ <anchored sort> ∧
parentAS0ofKind(*sort1*, <data type definition>)=*parentAS0ofKind*(*sort2*,<data type definition>))∨
(*sort1*∈ <expanded sort> ∧ *sort2*∈ <expanded sort> ∧
getEntityDefinition$_0$(*sort1*.**s**-<basic sort>, **type**).*derivedDateType*$_0$ =
getEntityDefinition$_0$(*sort2*.**s**-<basic sort>, **type**).*derivedDateType*$_0$)∨
(*sort1*∈ <reference sort> ∧ *sort*2∈ <reference sort> ∧
getEntityDefinition$_0$(*sort1*.**s**-<basic sort>, **type**).*derivedDateType*$_0$ =
getEntityDefinition$_0$(*sort2*.**s**-<basic sort>, **type**).*derivedDateType*$_0$)∨
(*sort1*∈ <pid sort> ∧ *sort2*∈ <pid sort> ∧
getEntityDefinition$_0$(*sor*t1, **sort**) = *getEntityDefinition*$_0$(*sor*t2, **sort**))

Determine if two sort lists are the same.

isSameSortList$_0$(*sl*, *sl'*: <sort>*): *BOOLEAN* =$_{def}$
(*sl.length* = *sl'.length*) ∧
(∀*i*∈ 1.. *sl.length*: *isSameSort*$_0$(*sl*[*i*], *sl'*[*i*]))

The function *isDirectlySortCompatible*$_0$ is used to determine if the sort in the first argument is directly sort compatible to the one in the second.

isDirectlySortCompatible$_0$(*sort1*:<sort>, *sort2*:<sort>): *BOOLEAN* =$_{def}$
(*sort1*∈ <basic sort> ∧ *isObjectSort*$_0$(*sort2*) ∧ *isSubSort*$_0$(*sort1*, *sort2*)) ∨
(*sort1*∈ <anchored sort> ∧ *sort1*.**s**-<basic sort> = *sort2*) ∨
(*sort1* ∈ <reference sort> ∧ *sort1*.**s**-<basic sort> = *sort2*) ∨
(*sort2* ∈ <reference sort> ∧ *sort1*.**s**-<basic sort> = *sort1*) ∨
(*sort1* ∈ <expanded sort> ∧ *sort1*.**s**-<basic sort> = *sort2*) ∨
(*sort2* ∈ <expanded sort> ∧ *sort1*.**s**-<basic sort> = *sort1*) ∨
(*sort1* ∈ <pid sort> ∧ *isSubSort*$_0$(*sort1*, *sort2*))

The function *isObjectSort*$_0$ is used to determine if a sort is an object sort.

isObjectSort$_0$(*sort*: <sort>): *BOOLEAN* =$_{def}$
case *sort* **of**
| <basic sort> =>
let *dtd* = *getEntityDefinition*$_0$(*sort*, **type**).*derivedDateType*$_0$ **in**
if *dtd*.**s**-<data type heading>.**s**-<data type kind>=**object then** *True*
else *False*
endif
endlet
| <anchored sort> => *isObjectSort*$_0$(*sort*.**s**-<basic sort>)
| <expanded sort> => *True*
otherwise *False*
endcase

The function *isPidSort*$_0$ is used to determine if a sort is an pid sort.

isPidSort$_0$(*sort*: <sort>): *BOOLEAN* =$_{def}$
getEntityDefinition$_0$(*sort*,**sort**)∈ <interface definition>

The function *isSubSort*$_0$ is used to determine if the sort given in the first argument is a super sort of the one in the second.

isSubSort$_0$(*sort1*: <sort>, *sort2*: <sort>): *BOOLEAN* =$_{def}$
let *td1* = *getEntityDefinition*$_0$(*sort1*, **sort**) **in**
let *td2* = *getEntityDefinition*$_0$(*sort2*, **sort**) **in**

```
        (td1∈<interface definition>⇒isSubtype0(td1, td2)) ∧
        (td1∈<data type definition>∪<syntype definition>⇒
            isSubtype0(td1.derivedDateType0, td2.derivedDateType0))
    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, raises> =>
        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>(name1, name1, < specializeArgument(a, spec): a in arguments>,
                                         specializeArgument(result, spec), raise)
            endlet
    otherwise
        undefined
    endif
```

The function *specializeArgument* replaces every <anchored sort> with the specialized sort.

```
specializeArgument(arg: <argument> ∪ <result> ∪ <sort>, spec: <data type specialization>):
        <argument> ∪ <result> ∪ <sort> =def
    case arg in
    | <argument>(virt, <formal parameter>(kind, sort)) =>
        mk-<argument>(virt, mk-<formal parameter>(kind, specializeArgument(sort, spec)))
    | <result>(sort) => mk-<result>(specializeArgument(sort, spec))
    | <anchored sort>(*) => mk-<anchored sort>(spec.parentAS0.name0)
    otherwise
        arg
    endcase
```

2.9.5 Operations

Abstract Syntax

```
Static-operation-signature      =   Operation-signature
Dynamic-operation-signature     =   Operation-signature
Operation-signature             ::  Operation-name Formal-argument* [ Result ] Identifier

Formal-argument                 =   Virtual-argument | Nonvirtual-argument
Virtual-argument                ::  Argument
Nonvirtual-argument             ::  Argument
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> | <name class operation> }
<argument>* [<result>] <raises>

<operation preamble> :: [<virtuality>] [<visibility>]

<argument> ::[<argument virtuality>] <formal parameter>

<formal parameter> :: <parameter kind> <sort>

<argument virtuality> :: ()

<result> :: <sort>

Conditions on Concrete Syntax

$\forall os \in$ <operation signature> :
$os.parentAS0 \in$ <operator list> $\Rightarrow (\neg \exists v \in$ <virtuality> $\cup$ <argument virtuality>: $isAncestorAS0(os, v))$

If <operation signature> is contained in an <operator list>, then the <operation signature> must not contain <virtuality> or <argument virtuality>.

$\forall os \in$ <operation signature> :
$(os.parentAS0 \in$ <method list> $\wedge\ os.virtuality_0 = undefined) \Rightarrow$
$(\neg \exists av \in$ <argument virtuality>:$(isAncestorAS0(os, av)))$

If <operation signature> is contained in a <method list> and <virtuality> is not present, then none of the <argument>s must contain <argument virtuality>.

$\forall os \in$ <operation signature> : os.**s-**<result> = $undefined \Rightarrow os.parentAS0 \in$ <method list>

<result> in <operation signature> may be omitted only if the <operation signature> occurred in a <method list>.

$\forall\ os \in$ <operation signature> :
$(\exists av \in$ <argument virtuality>: $av.parentAS0.parentAS0 = os) \Rightarrow os.virtuality_0 \in$ {**virtual**, **redefined** }

<argument virtuality> is legal only if <virtuality> contained the keywords **virtual** or **redefined**.

Transformations

$o =$ <operation signatures>(<operator list>(*operations*),
<method list>(<operation signature>(*pre*, *name*, *args*, *result*, *raises*) $\frown$ *rest*)
=2=>
<operation signatures>(<operator list>(*operations* $\frown$
<operation signature>(*pre*, *name*,
<argument>(<argument virtuality>(), $parentAS0ofKind(o, SCOPEUNIT_0).identifier_0$),
args, *result*, *raises*))), <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 keyword virtual, if <virtuality> was present, 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>.

<argument>(*virt*, <formal parameter>(*undefined*, *s* = <anchored sort>(basicsort)))
=2=>
<argument>(*virt*, <formal parameter>(<argument virtuality>, *s*)))

If the <sort> of an <argument> is an <anchored sort>, the <argument> implicitly contains <argument virtuality>.

s = <operation signature>(*p* = <operation preamble>(**redefined**, *), *name*, *args*, *result*, *raises*)
=2=>
<operation signature>(*p*, *name*, updateVirtuality($s.superCounterpart_0$, *args*), *result*, *raises*)

If an <operation signature> contains the keywords redefined in <virtuality>, for every <argument> in the matching <operation signature> of the base type, if this <argument> (implicitly or explicitly) contains <argument virtuality>, then the corresponding <argument> in <operation signature> implicitly also contains <argument virtuality>.

<formal parameter>(*undefined*, *sort*) =2=> <formal parameter>(**in**, *sort*)

An <argument> without an explicit <parameter kind> has the implicit <parameter kind> in.

Mapping to Abstract Syntax

| <operation signatures>(*operators*, *) => *Mapping*(*operators*).*toSet*

| *os* = <operation signature>(*, *name*, *arguments*, *result*, *) =>
if { *v*∈ <virtuality>∪<argument virtuality>: *isAncestorAS0*(*os*, *v*)} = *empty* **then**
mk-*Static-operation-signature*(*Mapping*(*name*), *Mapping*(*arguments*), *Mapping*(*result*),
Mapping(*os.operatorProcedureName*))
else
mk-*Static-operation-signature*(*Mapping*(*name*), *Mapping*(*arguments*), *Mapping*(*result*),
Mapping(*os.operatorProcedureName*))
endif

| <argument>(*undefined*, <formal parameter>(*, *name*)) => **mk-***Nonvirtual-argument*(*Mapping*(*name*))

| <argument>(*, <formal parameter>(*, *name*)) => **mk-***Virtual-argument*(*Mapping*(*name*))

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.

$operationSignatureParameterList_0$(*os*: <operation signature>):<argument>* =$_{def}$
(*os*.**s-**<argument>**-seq**)

Update the virtuality in the arguments list of the derived operation signature given the argument list of the base signature.

updateVirtuality(superargs: <argument>*, args: <argument>*):<argument> * =$_{def}$
let *arg1 = superargs.head* **in**
let *arg2 = args.head* **in**
< <argument>(args1.s-<argument virtuality>.s-<formal parameter>) > ⌢
updateVirtuality(*superargs.tail*, *args.tail)*

2.9.6 Any

Transformations

<data type definition>(*use*, *preamble*, *heading*, *undefined*, *body*)
=1=>
<data type definition>(*use*, *preamble*, *heading*,
<data type specialization>(<identifier>(<>, <name>("Any")),<renaming>(<>))))

If a <data type definition> does not contain an <data type specialization>, this is a shorthand notation for a <data type definition> with a <data type specialization>.

inherits Any;

2.9.7 Data Type Constructors

Concrete Syntax

<data type constructor> = <literal list> | <structure definition> | <choice definition>

2.9.7.1 Literals

Abstract Syntax

Literal-signature = *Single-literal* | *Nameclass-literal*

Single-literal :: *Literal-name Result*

Concrete Syntax

<literal list> :: [<visibility>] <literal signature>+

<literal signature> = <literal name> | <name class literal> | <named number>

<named number> :: <literal name> <Natural<simple expression>

Conditions on Concrete Syntax

$\forall$*num1*, *num2*$\in$ <named number>: *num1*. *parentAS0*=*num2*. *parentAS0*$\wedge$*num1*$\neq$*num2*$\Rightarrow$
num1.**s-**<simple expression>.$value_0 \neq$ *num2*.**s-**<simple expression>.$value_0$

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* $\frown$ < *name* > $\frown$ *tail*)
provided *name* $\in$ <literal name>
$\wedge \forall$ *n* $\in$ <literal name>: $\neg$ *n* **in** *head*
=8=>
<literal list>(*head* $\frown$ <named number>(*name*, *nextNumber*(*head*)) $\frown$ *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 $\neg$ *b.parentAS0.identifier*$_0$*.implicitSignaturesAdded*
=2=>
let *sort* = b.*parentAS0.identifier*$_0$ **in**
let *newoperators* =
<
<operation signature>(<operation preamble>(*undefined*, **public**), <name>("<"),
< <argument>(*undefined*, <formal parameter>(**in**, <anchored sort>(*sort*)),
<argument>(*undefined*, <formal parameter>(**in**, <anchored sort>(*sort*)) >,
<result>(<identifier>(<qualifier>(<name>("Predefined")), "Boolean")), <>),

<operation signature>(<operation preamble>(*undefined*, **public**), <name>(">"),
< <argument>(*undefined*, <formal parameter>(**in**, <anchored sort>(*sort*)),
<argument>(*undefined*, <formal parameter>(**in**, <anchored sort>(*sort*)) >,

<result>(<identifier>(<qualifier>(<name>("Predefined")), "Boolean")), <>),

<operation signature>(<operation preamble>(*undefined*, **public**), <name>(">="),
< <argument>(*undefined*, <formal parameter>(**in**, <anchored sort>(*sort*)),
<argument>(*undefined*, <formal parameter>(**in**, <anchored sort>(*sort*)) >,
<result>(<identifier>(<qualifier>(<name>("Predefined")), "Boolean")), <>),

<operation signature>(<operation preamble>(*undefined*, **public**), <name>("<="),
< <argument>(*undefined*, <formal parameter>(**in**, <anchored sort>(*sort*)),
<argument>(*undefined*, <formal parameter>(**in**, <anchored sort>(*sort*)) >,
<result>(<identifier>(<qualifier>(<name>("Predefined")), "Boolean")), <>),

<operation signature>(<operation preamble>(*undefined*, **public**), <name>("first"),
< >,
<result>(<anchored sort>(*sort*)), <>),

<operation signature>(<operation preamble>(*undefined*, **public**), <name>("last"),
< >,
<result>(<anchored sort>(*sort*)), <>),

<operation signature>(<operation preamble>(*undefined*, **public**), <name>("pred"),
< <argument>(*undefined*, <formal parameter>(**in**, <anchored sort>(*sort*))) >,
<result>(<anchored sort>(*sort*)), <>),

<operation signature>(<operation preamble>(*undefined*, **public**), <name>("succ"),
< <argument>(*undefined*, <formal parameter>(**in**, <anchored sort>(*sort*))) >,
<result>(<anchored sort>(*sort*)), <>),

<operation signature>(<operation preamble>(*undefined*, **public**), <name>("num"),
< <argument>(*undefined*, <formal parameter>(**in**, <anchored sort>(*sort*)),
<argument>(*undefined*, <formal parameter>(**in**, <anchored sort>(*sort*)) >,
<result>(<identifier>(<qualifier>(<name>("Predefined")), "Natural")), <>) >

in

<data type definition>(*entities*, *s*, <operations>(<operation signatures>(
<operator list>(*operators* ⌢ *newoperators*), *methods*), *refs*, *defs*), *init*)

endlet

and

b.parentAS0.identifier$_0$*.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>s in a <data type definition> are nominated in ascending order of the <Natural simple expression>s. 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.

If <literal signature> is a <regular expression>, this is shorthand for enumerating a (possibly infinite) set of <literal name>s as described in 12.1.9.1.

Mapping to Abstract Syntax

| <literal list>(*,*signatures*) => *Mapping*(*signatures*)

| <named number>(*name*, *number*) => **mk-***Literal-name*(*Mapping*(*name*), *Mapping*(*number*))

Auxiliary Functions

The functions $visibility_0$ gets the <visibility> part for an <operation signature> or a <literal signature>.

visibility$_0$(*s*:<operation signature>∪<literal signature>):<visibility>=$_{def}$
if *s*∈ <operation signature> **then** *s*.**s-**<operation preamble>.**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>(<>, <name>("0"))
elseif *literals*.*tail*.*nextNumber* ≠ *undefined* **then**
literals.*tail*.*nextNumber*
elseif literals.*head* ∈ <named number> **then**
<operator application>(<operation identifier>(<>, <name>("+")),
< *literals*.*head*.**s-**<simple expression>,
mk-<expression gen primary>(*undefined*, **mk-**<identifier>(<>, <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*
initially ∀ *id* ∈ <identifier>: *id*.*implicitSignaturesAdded* = *False*

2.9.7.2 Structure Data Types

Concrete Syntax

<structure definition> :: [<visibility>] <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>] <field<name>+ <sort>

Conditions on Concrete Syntax

$\forall sd \in$ <structure definition>: $sd.fieldNameList_0.length = |sd.fieldNameList_0.toSet|$

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*, < *s* >, *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*, < *s* >, *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*), <method list>(*methods*)), *refs*, *defs*), *init*)
    provided ¬ *b.parentAS0.identifier*$_0$*.implicitSignaturesAdded*
=5=>
    let *sort* = b.*parentAS0.identifier*$_0$ **in**
    let *newoperators* =
        <
            <operation signature>(<operation preamble>(*undefined*, **public**), <name>("Make"),
                < <argument>(*undefined*, <formal parameter>(**in**,*s1*)) | *s1* **in** *s.fieldSortList*$_0$>,
                <result>(*sort*), <>)
        >
    in let *newmethods* =
        < <operation signature>(<operation preamble>(**virtual**, **public**),
                <name>(*fields.fieldNameList*$_0$[*n*].s-*TOKEN* + "Modify"),
                < <argument>(*undefined*, <formal parameter>(**in**, *fields.fieldSortList*$_0$[*n*])) >,
                <result>(*sort*), <>) | *n* **in** 1..*fields.fieldNameList*$_0$*.length* > ⌢

        < <operation signature>(<operation preamble>(**virtual**, **public**),
                <name>(*fields.fieldNameList*$_0$[*n*].s-*TOKEN* + "Extract"),
                < >,
                <result>(*fields.fieldSortList*$_0$[*n*]), <>) | *n* **in** 1..*fields.fieldNameList*$_0$*.length* > ⌢

```
        <   <operation signature>(<operation preamble>(undefined, public),
                <name>(n.s-TOKEN + "Present"),
                < >,
                <result>(<identifier>(<qualifier>(<name>("Predefined")), "Boolean")), <>)
            | n in fields.fieldNameList0: n.isOptionalField0 >
    in
        <data type definition>(entities, s, <operations>(<operation signatures>(
            <operator list>(operators ∩ newoperators), <method list>(methods ∩ newmethods),
            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 (see D.3.16) 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 raise 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
=8=> <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
=2=>
    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.refersto*$_0$, {<data type definition>}.*qualifier*$_0$,
<name>(*cons.fieldNameList*$_0$[first.*length* + 1].s-*TOKEN* + "Modify")),
< *cons.fieldNameList*$_0$[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

The function *fieldNameList*$_0$ gets the list of field names defined in a <structure definition> or a <choice definition>.

fieldNameList$_0$(*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

The function *fieldSortList*$_0$ gets the list of field sorts defined in a <structure definition> or a <choice definition>.

fieldSortList$_0$(*sd*: <structure definition>∪<choice definition>):<sort>*=$_{def}$
<*fn.parentAS0*.**s-**<sort> | *fn* **in** *sd.fieldNameList*$_0$>

isOptionalField$_0$(*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

2.9.7.3 Choice Data Types

Concrete Syntax

<choice definition> :: [<visibility>] <choice of sort>*

<choice of sort> :: [<visibility>] <field<name> <sort>

Conditions on Concrete Syntax

∀*cd*∈ <choice definition>: *cd.fieldNameList*$_0$.*length* = |*cd.fieldNameList*$_0$.*toSet*|

Each <field name> of a choice sort must be different from every other <field name> of the same <choice definition>.

Transformations

let *nn*= *newName* **in**
< *d* = <data type definition>(*uses*, *preamble*, <data type heading>(*k*, *n*, *params*, *virt*), *spec*,
<data type definition body>(*entities*, <choice definition>(*visi*, *choices*), *ops*, *ini*)) >
=2=>
(**let** *emptyOperations* =
<operations>(<operation signatures>(<operator list>(*empty*), <method list>(*empty*)), *empty*, *empty*)

```
in
let anonStruct =
        <data type definition>(uses, undefined,
            <data type heading>(value, nn ^ "Struct", params, virt), spec,
                <data type definition body>(entities, <structure definition>(visi,
                        < <optional field>(<fields of sort>(c.s-<visibility>, < c.s-<name> >, c.s-<sort>))
                        | c in choices >),
                    ops, ini))
in
let anonLiterals =
        <data type definition>(empty, undefined,
            <data type heading>(value, nn ^ "Present", empty, undefined), undefined,
                <data type definition body>(empty, <literal list>(undefined,
                        < < c.s-<name> | c in choices >),
                    emptyOperations, undefined))
in
let anonChoice =
        <data type definition>(empty, undefined,
            <data type heading>(k, nn ^ "Choice", empty, undefined), undefined,
                <data type definition body>(empty, <structure definition>(undefined,
                    < <mandatory field>(<fields of sort>(protected, < <name>("Present") >,
                            <identifier>(d.fullQualifier0, nn ^ "Present"))),
                      <mandatory field>(<fields of sort>(protected, < <name>("Choice") >,
                            <identifier>(d.fullQualifier0, nn ^ "Struct")))
                      >),
                    emptyOperations, undefined))
in
let addOps =
    < <operation signature>(<operation preamble>(undefined, undefined), c.s-<name>,
            < <argument>(undefined, c.s-<sort>) >, <result>(d.identifier0), empty) | c in choices >
in
let addMethods =
    < <operation signature>(<operation preamble>(virtual, undefined), c.s-<name> ^ "Modify",
            < <argument>(undefined, c.s-<sort>) >, <result>(d.identifier0), empty) | c in choices > ^
    < <operation signature>(<operation preamble>(virtual, undefined), c.s-<name> ^ "Extract",
            empty, <result>(c.s-<sort>), empty) | c in choices > ^
    < <operation signature>(<operation preamble>(undefined, undefined), c.s-<name> ^ "Present",
            empty, <result>("Boolean"), empty) | c in choices >
in
let presentOp =
    <operation signature>(<operation preamble>(undefined, undefined), "PresentExtract",
            < <argument>(undefined, d.identifier0) >,
            <result>(<identifier>(d.fullQualifier0, nn ^ "Present")), empty)
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",
                    < <literal 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", < <literal identifier>(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
let realChoice =
        <data type definition>(uses, preamble,
            <data type heading>(k, n, empty, virt),
                <data type specialization>(
                    <type expression>(<identifier>(d.fullQualifier0, nn ^ "Choice"), empty),
                    <renaming>(< <rename pair gen operation name>(nn ^ "Make", "Make"),
                            <rename pair gen operation name>(nn ^ "PresentModify", "PresentModify"),
                            <rename pair gen operation name>(nn ^ "PresentExtract", "PresentExtract"),
                            <rename pair gen operation name>(nn ^ "ChoiceModify", "ChoiceModify"),
                            <rename pair gen operation name>(nn ^ "ChoiceExtract", "ChoiceExtract") >)),
                <data type definition body>(empty, undefined,
                    <operations>(
                        <operation signatures>(
                            <operator list>(addOps),
                            <method list>(addMethods ^ < presentOp >)),
                        empty,
                        addOpDefs ^ addMethodDefs ^ < presentOpDef >), undefined))
in

        < anonStruct, anonLiterals, anonChoice, realChoice >
```

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) A <data type definition> with an anonymous name, anonChoice, is constructed as follows:

```
object type anonChoice
struct
        protected Present anonPresent;
        protected Choice anon;
endobject type anonChoice;
```

if the original data type definition had defined on object sort. Otherwise, the <data type definition> is a <value data type definition>.

d) A <data type definition> is constructed as follows:

```
object type Choice-name inherits anonChoice (anonMake = Make,
                                anonPresentModify = PresentModify,
                                anonPresentExtract = PresentExtract,
                                anonChoiceModify = ChoiceModify,
                                anonChoiceExtract = ChoiceExtract )
adding
        operations
endobject type Choice-name;
```

if the original data type definition had defined on object type, and where operations is <operations>, as defined below. Otherwise, the <data type definition> is a <value data type definition>. The <renaming> renames the mentioned operations inherited from anonChoice to anonymous names.

e) 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.

f) 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 of 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.

g) 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.

2.9.8 Behaviour of Operations

Concrete Syntax

<operation definitions> =
    <operation definition> | <external operation definition>

<textual 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> | <statement list> }

<operation heading> ::
        <operation kind> <operation preamble> <qualifier> <operation name>
        <formal operation parameter>* [<operation result>] <raises>

<operation kind> = **operator** | **method**

<operation identifier> :: <qualifier> <operation name>

<formal operation parameter> ::
        [<argument virtuality>] <parameter kind> <parameters of sort>

<entity in operation> =
        <data definition>
    | <variable definition>
    | <exception definition>
    | <select definition>

<operation body> ::
        [<on exception>] <start> {<free action> | <exception handler>}*

<operation result> :: [<variable<name>] <sort>

Conditions on Concrete Syntax

$\forall$*opRef*$\in$ <textual operation reference>:
    (*opRef*.**s-**<operation heading> .**s-**<formal operation parameter>**-seq** = *empty* $\vee$
    opRef.**s-**<operation heading> .**s-**<operation result> = *undefined*)$\Rightarrow$
        ($\forall$*opRef1*$\in$ <textual operation reference>:
            opRef1 $\neq$ *opRef* $\wedge$ *opRef.parentAS0*= *opRef1.parentAS0*$\Rightarrow$ *opRef1.name*$_0$$\neq$ *opRef.name*$_0$)

$\forall$*opDef*$\in$ <external operation definition>:
    (*opDef*.**s-**<operation heading> .**s-**<formal operation parameter>**-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.name*$_0$$\neq$ *opDef.name*$_0$)

<formal operation parameters> and <operation result> in <textual operation reference> and <external operation definition> may be omitted if there is no other <textual operation reference> or <external operation definition>, respectively, within the same sort which has the same name.

$\forall$*od*$\in$ <operation definition>: $\exists$*os*$\in$ <operation signature> :
    od.parentAS0= *os.parentAS0*$\wedge$ *od.name*$_0$= *os.name*$_0$$\wedge$ *isCompatibleOperationAndSignature*$_0$(*od*, *os*)

For each <operation definition> there must exist an <operation signature> in the same scope unit having the same <operation name>, positionally having the same <argument sort>s and <parameter kind>s as specified in the <formal operation parameters> (if present) and having the same <result sort> as specified in <operation result> (if present).

$\forall os \in$ <operation signature> : $\exists! od \in$ <operation definition>:
$od.parentAS0 = os.parentAS0 \wedge od.name_0 = os.name_0 \wedge isCompatibleOperationAndSignature_0(od, os)$

For each <operation signature> at most one corresponding <operation definition> can be given.

$\forall bs \in$ <operation body>$\cup$<statement>: $parentAS0ofKind(bs,$ <operation definition>$) \neq undefined \Rightarrow$
$(\neg \exists ie \in$ <imperative expression>: $isAncestorAS0(bs, ie)) \wedge$
$(\forall id \in$ <identifier>: $id.idKind_0 \notin \{$**synonym**, **procedure**$\} \wedge isAncestorAS0(bs, id) \Rightarrow$
$isDefinedIn_0(getEntityDefinition_0(id, id.idKind_0), parentAS0ofKind(bs,$ <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

od = <operation definition>(*, <operation heading>(*kind*, *preamble*, *, *name*, *params*, *result*, *raises*) , *, *)
provided $\neg \exists$ *os* $\in$ <operation signature>: $isCompatibleOperationAndSignature_0(od, os)$
$\wedge$ kind = **method**
=2=> *od*
and
od.parentAS0.**s-**<operation signatures>.**s-**<method list>.**s-**<operation signature>**-seq**
=>
od.parentAS0.**s-**<operation signatures>.**s-**<method list>.**s-**<operation signature>**-seq** $\frown$
< <operation signature>(*preamble*, *name*, *params*, *result*, *raises*) >

od = <operation definition>(*, <operation heading>(*kind*, *preamble*, *, *name*, *params*, *result*, *raises*) , *, *)
provided $\neg \exists$ *os* $\in$ <operation signature>: $isCompatibleOperationAndSignature_0(od, os)$
$\wedge$ kind = **operator**
=2=> *od*
and
od.parentAS0.**s-**<operation signatures>.**s-**<operator list>.**s-**<operation signature>**-seq**
=>
od.parentAS0.**s-**<operation signatures>.**s-**<operator list>.**s-**<operation signature>**-seq** $\frown$
< <operation signature>(*preamble*, *name*, *params*, *result*, *raises*) >

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*), *raises*), *entities*, *body*)
provided *od.operatorProcedureName* = *undefined*
=5=>
od
and
od.getEntities
=> *od.getEntities* $\frown$
< <procedure definition>(*use*,
<procedure heading>(*undefined*, *undefined*, *nn*, <>, *undefined*, *undefined*,
(**if** *kind* = **method then**
< <formal procedure parameter>(**inout**, <parameters of sort>(< thisname >,
parentAS0ofKind(od, <data type definition>).$identifier_0$)) >
else < > **endif**) $\frown$
< <formal procedure parameter>(*p*.**s-**<parameter kind>, *p*.**s-**<parameters of sort>) |
p **in** *params* >,
<procedure result>(*var*, *sort*), *raises*),
entities, *makeProcedureBody*(*body*))>
and
od.operatorProcedureName := *nn*

An <operation definition> is transformed into a <procedure definition> or <procedure diagram> respectively, having 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 <statement list>, the result of this transformation is a <procedure definition> (see 9.4). After the Model of <procedure definition> has been applied, the virtual start inserted by that Model is replaced by a start without <virtuality>.

An <operation diagram> is transformed into a <procedure diagram> in a similar manner.

The Procedure-definition corresponding to the resultant <procedure definition> or <procedure diagram> is associated with the Operation-signature represented by the <operation signature>.

If the <operation definition> or <operation diagram> defines a method, then during the transformation into a <procedure definition>, or <procedure diagram> 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 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.

Auxiliary Functions

The functions *isCompatibleOperationAndSignature*$_0$ determines if an <operation definition> and an <operation signature> are compatible.

isCompatibleOperationAndSignature$_0$(*od*: <operation definition>, *os*: <operation signature>): *BOOLEAN*=$_{\text{def}}$
let *seq1* = *od.operationFormalparameterList*$_0$ **in**
let *seq2* = *os.operationSignatureParameterList*$_0$ **in**
(*od*.**s-**<operation heading> .**s-**<operation result> ≠ *undefined* ⇒
isSameResult$_0$(*od*.**s-**<operation heading> .**s-**<operation result> , *os*.**s-**<result>)∧
(*seq1*≠*empty* ⇒
seq1.length = *seq2.length*∧
(∀*i*∈ 1..*seq1.length*: *isSameSort*$_0$(*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.

operationFormalparameterList$_0$(*od*: <operation definition>): <name>* =$_{\text{def}}$
< *opl*.**s-**<parameters of sort>.**s-**<<name>**-seq** |
opl **in** *od*.**s-**<operation heading>.**s-**<formal operation parameter>**-seq** >

The following determines the entity kind of an <identifier> according to its position.

idKind$_0$(*i*: <identifier>): *ENTITYKIND*$_0$ =$_{\text{def}}$
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**
| <signal reference> => **signal**
| <data type reference> => **type**
| <textual interface gate definition> => **interface**
| <textual endpoint constraint> => *parentAS0ofKind*(*i*, *TYPEDEFINITION*$_0$).*kind*$_0$
| <agent type context parameter> => **agent type**
| <agent constraint gen atleast> => **agent type**
| <procedure context parameter> => **procedure**
| <signal context parameter> => **signal**
| <composite state type context parameter> => **state type**
| <interface constraint> => **interface**
| <virtuality constraint> => *parentAS0ofKind*(*i*, *TYPEDEFINITION*$_0$).*kind*$_0$
| <procedure preamble> => **remote procedure**
| <raises> => **exception**

| <channel endpoint> =>
if *getEntityDefinition*$_0$(*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 *getEntityDefinition*$_0$(*i*, **signal**) ≠ *undefined* **then signal**
elsif *getEntityDefinition*$_0$(*i*, **signallist**) ≠ *undefined* **then signallist**
elsif *getEntityDefinition*$_0$(*i*, **timer**) ≠ *undefined* **then timer**
elseif *getEntityDefinition*$_0$(*i*, **remote procedure**) ≠ *undefined* **then remote procedure**
elseif *getEntityDefinition*$_0$(*i*, **remote variable**) ≠ *undefined* **then remote variable**
else interface
endif
| <remote procedure call body> => **remote procedure**
| <import expression> => **remote variable**
| <export> => **variable**
| <entry point> => **state**
| <exit point> => **state**
| <raise body> => **exception**
| <create body> =>
if *getEntityDefinition*$_0$(*i*, **agent**) ≠ *undefined* **then agent**
else agent type
endif
| <procedure call body> => **procedure**
| <output body> => **signal**
| <via path> =>
if *getEntityDefinition*$_0$(*i*, **channel**) ≠ *undefined* **then channel**
else gate
endif
| <destination> => **timer**
| <loop variable indication> => **variable**
| <set clause> => **timer**
| <reset clause> => **timer**
| <exception stimulus> => **exception**
| <interface reference> => **interface**
| <range check expression gen identifier> => **sort**
| <variables of sort gen name> => **remote variable**
| <timer active expression> => **timer**
| <type expression> => *parentAS0ofKind*(*i*, *TYPEDEFINITION*$_0$).*kind*$_0$
| *a*=<actual context parameter> =>
take({*f* ∈ <formal context parameter>: *isContextParameterCorresponded*$_0$(*a*, *f*) }).*entityKind*$_0$
| <communication constraint> => **timer**
| <anchored sort> ∪ <expanded sort> ∪ <reference sort> ∪ <formal parameter>
∪ <variable context parameter> ∪ <remote variable context parameter> ∪ <sort constraint>
∪ <parameters of sort> ∪ <procedure result> ∪ <remote variable definition>
∪ <local variables of sort> ∪ <loop variable definition> ∪ <interface variable definition>
∪ <result> ∪ <field> ∪ <formal operation parameter> ∪ <operation result>
∪ <internal synonym definition item> ∪ <external synonym definition item>
∪ <range check expression> ∪ <variables of sort> ∪ <any expression>
∪ <synonym context parameter> ∪ <syntype definition gen syntype> => **sort**
| <method application> => **method**
| <operator application> => **operator**
| <indexed primary> ∪ <field primary> ∪ <actual context parameter> ∪ <expression gen primary> =>
if *getEntityDefinition*$_0$(*i*, **variable**) ≠ *undefined* **then variable**
elseif *getEntityDefinition*$_0$(*i*, **synonym**) ≠ *undefined* **then synonym**
else literal
endif
| <stimulus> => **variable**
| <exception stimulus> => **variable**
| <assignment> => **variable**
| <indexed variable> => **variable**

| <field variable> => **variable**
endcase

makeProcedureBody(b: <operation body> ∪ <statement list>) <procedure body> ∪ <statement>* =$_{def}$
case *b* **of**
| <operation body>(*onexc*, *start*, *actions*) => <procedure body>(*onexc*, *start*, *actions*)
| <statement list>(*, *) => <compound statement>(*b*)
otherwise
undefined
endcase

2.9.9 Additional Data Definition Constructs

2.9.9.1 Name Class

Abstract Syntax

Nameclass-literal	::	*Regular-Expression**
Regular-Expression	::	*Partial-regular-expression**
Partial-regular-expression	::	**ANY**
Regular-element [*Regular-multiplicity*]		
Regular-multiplicity	=	*AGENTNAT*
	|	**ATLEASTONCE**
	|	**ANY**
Regular-element	=	*Nameclass-literal* | *TOKEN* | *Regular-interval*
Regular-interval	::	*TOKEN TOKEN*

Concrete Syntax

<name class operation> :: <operation<name> <regular expression>

<name class literal> :: <regular expression>

<regular expression> :: <or partial regular expression>+

<or partial regular expression> :: <partial regular expression>+

<partial regular expression> ::
<regular element> [<Natural<name> | <plus sign> | <asterisk>]

<regular element> =
<regular expression>
| <character string>
| <regular interval>

<regular interval> :: <character string> <character string>

Conditions on Concrete Syntax

∀*os*∈ <operation signature> : *os*.**s- s-implicit** ∈ <name class operation> ⇒
os. *parentAS0*∈ <operator list> ∧ *os*.**s-**<argument>**-seq** = *empty*

A <name class operation> can only be used in an <operation signature>. An <operation signature> containing <name class operation> must only occur in an <operator list> and must not contain <arguments>.

Transformations

This transformation is handled by the Mapping.

A <name class literal> is equivalent to this set of names in the abstract syntax. When a <name class operation> is used in an <operation signature>, a set of <operation signature>s is created by substituting each name in the equivalent set of names for the <name class operation> in the <operation signature>.

A <regular expression> which is a list of <partial regular expression>s without an or specifies that the names can be formed from the characters defined by the first <partial regular expression> followed by the characters defined by the second <partial regular expression>.

When an or is specified between two <partial regular expression>s, then the names are formed from either the first or the second of these <partial regular expression>s. or is more tightly binding than simple sequencing.

If a <regular element> is followed by <Natural literal name>, the <partial regular expression> is equivalent to the <regular element> being repeated the number of times specified by the <Natural literal name>.

If a <regular element> is followed by '*' the <partial regular expression> is equivalent to the <regular element> being repeated zero or more times.

If a <regular element> is followed by <plus sign> the <partial regular expression> is equivalent to the <regular element> being repeated one or more times.

A <regular element> which is a bracketed <regular expression> defines the character sequences defined by the <regular expression>.

A <regular element> which is a <character string> defines the character sequence given in the character string (omitting the quotes).

A <regular element> which is a <regular interval> defines all the characters specified by the <regular interval> as alternative character sequences. The characters defined by the <regular interval> are all the characters greater than or equal to the first character and less than or equal to the second character according to the definition of the Character sort (see D.2).

The names generated by a <name class literal> are defined in the alphabetical order according to the ordering of the character sort. The characters are considered case sensitive, and a true prefix of a word is considered less than the whole word.

NOTE – Examples can be found in Annex D.

Mapping to Abstract Syntax

| <name class literal>(*name*) => *Mapping*(*name*)
| <regular expression>(*ors*) => **mk-***Nameclass-literal*(*Mapping*(*ors*))
| <or partial regular expression>(*partials*) => **mk-***Regular-Expression*(*Mapping*(*partials*))

| <partial regular expression>(*element*, n=<name>(*)) =>
mk-*Partial-regular-expression*(*Mapping*(*element*), *Mapping*(*n*))
| <partial regular expression>(*element*, <plus sign>) =>
mk-*Partial-regular-expression*(*Mapping*(*element*), **ATLEASTONCE**)
| <partial regular expression>(*element*, <plus sign>) =>
mk-*Partial-regular-expression*(*Mapping*(*element*), **ANY**)

| <regular interval>(*contents*) => *Mapping*(*contents*)
| <regular interval>(*low*, *high*) => **mk-***Regular-interval*(*Mapping*(*low*), *Mapping*(*high*))

Auxiliary Functions

The function $isInRegularExpr_0$ is used to determine if a name is in the specified regular expression.

$isInRegularExpr_0$(*name*: <name>, *regExpr*: <or partial regular expression>*): *BOOLEAN* =$_{def}$
if *regExpr* = *empty* **then** *False*
else *isInPartialRegularExpression*(*name*, *regExpr.head*.**s-**<partial regular expression>**-seq**)
$\vee$ $isInRegularExpr_0$(*name*, *regExpr.tail*)
endif

The function *isInPartialRegularExpression* is true if the name matches the partial regular expression.

isInPartialRegularExpression(*name*: <name>, *regExpr*: <partial regular expression>**-seq**): *BOOLEAN* =$_{def}$
if *regExpr* = *empty* **then** *name* = *empty*

else ∃ *name1*, *name2* ∈ <name>: *name1* ⌢ *name2* = *name* ∧
isInPartialRegularExpression(*name2*, *regExpr.tail*) ∧
case *regExpr.head*.**s-implicit of**
| <name> => *regularExpressionRepeated*(*name2*,
value$_0$(*regExpr.head*.**s-implicit**), *regExpr.head*.**s-**<regular element>))
| <plus sign> => ∃ *name21*, *name22* ∈ <name>: *name21* ⌢ *name22* = *name2* ∧
regularElementMatches(*name21*, *regExpr.head*.**s-**<regular element>) ∧
asteriskMatches(*name22*, *regExpr.head*.**s-**<regular element>))
| <asterisk> => *asteriskMatches*(*name2*, *regExpr.head*.**s-**<regular element>))
endcase
endif

The function *regularExpressionRepeated* is true if name matches exactly count times to regExpr.

regularExpressionRepeated(*name*: <name>, *count*: *NAT*, *regExpr*: <regular element>): *BOOLEAN* =$_{def}$
if *count* = 0 **then**
name = *empty*
elseif *count* = 1 **then**
regularElementMatches (*name*, *regExpr*)
else
(∃ *name1*, *name2* ∈ <name>: *name1* ⌢ *name2* = *name* ∧
regularElementMatches (*name1*, *regExpr*)
∧ *regularExpressionRepeated*(*name2*, *count* - 1, *regExpr*))

The function *asteriskMatches* is true if the regular expression repeated any number of times matches the name.

asteriskMatches(*name*: <name>, *regExpr*: <regular element>)=$_{def}$
if *name* = *empty* **then**
True
else
(∃ *name1*, *name2* ∈ <name>: *name1* ⌢ *name2* = *name* ∧
regularElementMatches (*name1*, *regExpr*)
∧ *asteriskMatches*(*name2*, *regExpr*))

The function *regularElementMatches* is true if regExpr matches name.

regularElementMatches(*name*: <name>, *regExpr*: <regular element>) =$_{def}$
if *regExpr* ∈ <regular expression> **then**
isInRegularExpr$_0$(*name*,*regExpr*.**s-**<regular expression>.**s-**<or partial regular expression>**-seq**)
elseif *regExpr* ∈ <character string> **then**
name = *regExpr*
else
name.length=1 ∧ *regExpr*.**s-**<character string>[1] ≤ *name*[1] ∧
name[1] ≤ *regExpr*.**s-**<character string>[1]
endif

The function determines if the specified two <name class literal>s are the same.

isSameNameClassLiteral$_0$(*ncl1*:<name class literal>, *ncl2*:<name class literal>): *BOOLEAN* =$_{def}$
∀*n*∈ <name>:
(*isInRegularExpr*$_0$(*n*, *ncl1*) ⇒*isInRegularExpr*$_0$(*n*, *ncl2*))∧
(*isInRegularExpr*$_0$(*n*, *ncl2*) ⇒*isInRegularExpr*$_0$(*n*, *ncl1*))

isSameNameClassOperation$_0$(*ncl1*:<name class operation>, *ncl2*:<name class operation>): *BOOLEAN* =$_{def}$
ncl1.**s-**<name>= *ncl2*.**s-**<name>

2.9.9.2 Name Class Mapping

Concrete Syntax

<spelling term> :: <operation><name>

Transformations

op=<operator application>(*identifier*, *parameters*)
provided ∃ op ∈ <name class operation>:
resolutionByContext$_0$(identifier) = *op*
∧ ¬ *identifier.implicitSignaturesAdded*
=8=>
op
and
let *origop* = *resolutionByContext*$_0$(*identifier*) **in**
let *ops* = *take*({*op1* ∈ <operations>: *origop.parentAS0.parentAS0* = *op1*}) **in**
ops.**s-**<operation definitions>**-seq**
=8=>
ops.**s-**<operation definitions>-seq ⌢
< *replaceInSyntaxTree*(<spelling term>(*identifer*.s-<name>), <literal>(*identifer*.s-<name>), *origop*) >

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 through 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 (see D.3.4).

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>.

2.9.9.3 Restricted Visibility

Concrete Syntax

<visibility> = **public** | **protected** | **private**

Conditions on Concrete Syntax

∀*d*∈ <data type definition>: *d.specialization*$_0$≠ *undefined* ⇒
(∀*cons*∈ <data type constructor>: *cons.surroundingScopeUnit*$_0$ = *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.virtuality$_0$∈ {**redefined**, **finalized**}⇒*os.visibility*$_0$=*undefined*

<visibility> must not be used in an <operation signature> that redefines an inherited operation signature.

Transformations

<operation preamble>(*virt*, **public**) **provided** *vis* ≠ *undefined* =1=> <operation preamble>(*virt*, *undefined*)

<literal list>(**public**, *sigs*) **provided** *vis* ≠ *undefined* =1=> <literal list>(*undefined*, *sigs*)

<structure definition>(**public**, *fields*) **provided** *vis* ≠ *undefined* =1=> <structure definition>(*undefined*, *fields*)

<fields of sort>(**public**, *fields*, *sort*) **provided** *vis* ≠ *undefined* =1=> <fields of sort>(*undefined*, *fields*, *sort*)

<choice definition>(**public**, *fields*) **provided** *vis* ≠ *undefined* =1=> <choice definition>(*undefined*, *fields*)

<choice of sort>(**public**, *fields*, *sort*) **provided** *vis* ≠ *undefined* =1=> <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 $isPrivate_0$ determines if a <literal signature> or an <operation signature> is private.

$isPrivate_0$(*s*: <literal signature> ∪ <operation signature>): *BOOLEAN* $=_{def}$
($s.visibility_0$ = **private**)

The function $isPublic_0$ determines if a <literal signature> or an <operation signature> is public.

$isPublic_0$(*s*:<literal signature> ∪ <operation signature>): *BOOLEAN* $=_{def}$
($s.visibility_0 \neq$ **private**) ∧ ($s.visibility_0 \neq$ **protected**)

2.9.9.4 Syntypes

Abstract Syntax

Syntype-definition	::	*Syntype-name* *Parent-sort-identifier* *Range-condition*
Parent-sort-identifier	=	*Sort-identifier*

Concrete Syntax

<syntype> = <syntype><identifier>

<syntype definition> ::
<package use clause>* { <syntype definition gen syntype> | <syntype definition gen type preamble> }

<syntype definition gen syntype> ::
<syntype><name> <parent sort identifier>
[<default initialization>] [<constraint>]

<syntype definition gen type preamble> ::
<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 gen type preamble>(*preamble*,
<data type heading>(*kind*, *name*, *params*, *vconstr*) , *spec*, *body*, *constr*)) >
=8=>
< <syntype definition>(*uses*,
<syntype definition gen syntype>(name, <identifier>(<>, 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 gen syntype>(*name*, *parent*, *, *constr*)) =>
mk-*Syntype-definition*(*Mapping*(*name*), *Mapping*(*parent*), *Mapping*(*constr*))

2.9.9.5 Constraint

Abstract Syntax

Range-condition	::	*Condition-item*-**set**
Condition-item	=	*Open-range*
	|	*Closed-range*
	|	*Size-range*
Open-range	::	*Operation-identifier Constant-expression*
Closed-range	::	*Open-range Open-range*
Size-range	::	*Range-condition*

Concrete Syntax

<constraint> = <range condition> | <size constraint>

<range condition> = <range>+

<range> = <closed range> | <open range>

<open range> = <constant> | <open range gen greater than or equals sign>

<open range gen greater than or equals sign> ::
{ <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

$\forall sd \in$ <syntype definition>: *sd*.**s-implicit**.**s-**<constraint>$\in$ <range condition> $\Rightarrow$
(**let** *rc* = *sd*.**s-implicit**.**s-**<constraint> **in**
($\forall sym \in$ <less than sign>: *isAncestorAS0*(*rc*, *sym*)$\Rightarrow$ $isDefinedSym_0$(*sd*, "<"))$\wedge$
($\forall sym \in$ <greater than sign>: *isAncestorAS0*(*rc*, *sym*)$\Rightarrow$ $isDefinedSym_0$(*sd*, ">"))$\wedge$
($\forall sym \in$ <less than or equals sign>: *isAncestorAS0*(*rc*, *sym*)$\Rightarrow$ $isDefinedSym_0$(*sd*, "<="))$\wedge$
($\forall sym \in$ <greater than or equals sign>: *isAncestorAS0*(*rc*, *sym*)$\Rightarrow$ $isDefinedSym_0$(*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).

$\forall sd \in$ <syntype definition>: *sd*.**s-implicit**.**s-**<constraint>$\in$ <range condition>$\Rightarrow$
$\forall cr \in$ <closed range>: *isAncestorAS0*(*sd*.**s-implicit**.**s-**<range condition>,*cr*)$\Rightarrow$ $isDefinedSym_0$(*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.

$\forall sd \in$ <syntype definition>:$\forall rc \in$ <range condition>:$\forall ce \in$ <constant expression>:
isAncestorAS0(*rc*, *ce*)$\wedge$$rc.surroundingScopeUnit_0$=*sd*$\Rightarrow$$isSameSort_0$($ce.staticSort_0$, $sd.identifier_0$)

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>⇒*isDefinedSym*$_0$(*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 gen greater than or equals sign>(<equals sign>(), *const*) =>
mk-*Open-range*(*rangeOperator*(r, "="), *Mapping*(*const*))
| *r* = <open range gen greater than or equals sign>(<not equals sign>(), *const*) =>
mk-*Open-range*(*rangeOperator*(r, "/="), *Mapping*(*const*))
| *r* = <open range gen greater than or equals sign>(<less than sign>(), *const*) =>
mk-*Open-range*(*rangeOperator*(r, "<"), *Mapping*(*const*))
| *r* = <open range gen greater than or equals sign>(<greater than sign>(), *const*) =>
mk-*Open-range*(*rangeOperator*(r, ">"), *Mapping*(*const*))
| *r* = <open range gen greater than or equals sign>(<less than or equals sign>(), *const*) =>
mk-*Open-range*(*rangeOperator*(r, "<="), *Mapping*(*const*))
| *r* = <open range gen greater than or equals sign>(<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*))
| <size constraint>(range) => { **mk-***Size-range*(*Mapping*(*range*)) }

Auxiliary Functions

The function *isPredefSort*$_0$ is used to determine the predefined sorts.

isPredefSort$_0$(*s*: <sort>) : *BOOLEAN* =$_{\text{def}}$
getEntityDefinition$_0$(*s*, **sort**)=*undefined*∧*s*.**s-**<name>∈ *PREDEFINEDSORT*$_0$

The function *isDefinedSym*$_0$ 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.

isDefinedSym$_0$(*sd*:<syntype definition>, *sym*: *SYMBOL*$_0$): *BOOLEAN* =$_{\text{def}}$
(**let** *dtd*=*sd.derivedDateType*$_0$ **in**
if *sym*∈ {"<",">","<=",">="} **then**
(∃*ll*∈ <literal list>: *ll.surroundingScopeUnit*$_0$ = *dtd*)∨
(∃*os*∈ <operation signature>: (*os.surroundingScopeUnit*$_0$=*dtd*)∧
(**let** *fpl*= *os.operationSignatureParameterList*$_0$ **in**
os.entityName$_0$= *sym* ∧
isPredefSort$_0$(*os*.**s-**<result>) ∧ *os*.**s-**<result>.**s-**<name>= "Boolean" ∧
fpl.length = 2 ∧
getEntityDefinition$_0$(*fpl*[1].**s-**<formal parameter>.**s-**<sort>, **sort**) =*sd*∧
getEntityDefinition$_0$(*fpl*[2].**s-**<formal parameter>.**s-**<sort>, **sort**) =*sd*
endlet))
else // *sym*∈ {"Length"}
(∃*os*∈ <operation signature> :*os.surroundingScopeUnit*$_0$=*dtd*∧
(**let** *fpl*=*os.operationSignatureParameterList*$_0$ **in**
os.name$_0$= "Length"∧
isPredefSort$_0$(*os*.**s-**<result>) ∧ *os*.**s-**<result>.**s-**<name>= "Natural"∧

fpl.length= 1∧
getEntityDefinition$_0$(*fpl*[1].**s-**<formal parameter>.**s-**<sort>, **sort**) *derivedDateType*$_0$=*dtd*
endlet))
endlet)

rangeOperator(t: *TOKEN*): *Identifier*

2.9.9.6 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>: ¬ *isContainedInConsExp*$_0$(*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>.*staticSortSet*$_0$∧*isSameSort*$_0$(*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>.*staticSortSet*$_0$|>1⇒ *sdi*.**s-**<sort> ≠ *undefined*

If the sort of the <constant expression> can not be uniquely determined, then a sort must be specified in the <synonym definition>.

Auxiliary Functions

The function *isContainedInConsExp*$_0$ is used to determine if a <constant expression> refer to the synonym defined by the enclosing <synonym definition> either directly or indirectly.

isContainedInConsExp$_0$(*def*: <internal synonym definition item>, *exp*: <constant expression>): *BOOLEAN* =$_{\text{def}}$
∃*synoId*∈ <synonym>: *isAncestorAS0*(*exp*, *synId*) ∧
(*def* = *getEntityDefinition*$_0$(*synoId*, **synonym**)∨
isContainedInConsExp$_0$(*def*, *getEntityDefinition*$_0$(*synoId*, **synonym**).**s-**<constant expression>))

2.9.10 Expression

2.9.10.1 Expression

Abstract Syntax

Expression	=	*Constant-expression*
	|	*Active-expression*
Constant-expression	=	*Literal*
	|	*Conditional-expression*
	|	*Equality-expression*

	|	*Operation-application*
	|	*Range-check-expression*
Active-expression	=	*Variable-access*
	|	*Conditional-expression*
	|	*Operation-application*
	|	*Equality-expression*
	|	*Imperative-expression*
	|	*Range-check-expression*
	|	*Value-returning-call-node*
Imperative-expression	=	*Now-expression*
	|	*Pid-expression*
	|	*Timer-active-expression*
	|	*Any-expression*

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> =
    <create expression>
  | <value returning procedure call>
  | <range check expression>
  | <binary expression>
  | <equality expression>
  | <expression gen primary>

<simple expression> = <constant expression>

<constant expression> = <constant<expression>

<binary expression> ::
    <expression>
    { <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** }
    <expression>

<expression gen primary> :: [<hyphen> | **not**] <primary>

<primary> =
    <operator application>
  | <literal>
  | <expression>
  | <conditional expression>
  | <spelling term>
  | <extended primary>
  | <active primary>
  | <synonym>

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

<imperative expression> =
    <now expression>
  | <import expression>
  | <pid expression>
  | <timer active expression>
  | <any expression>
  | <state expression>

Conditions on Concrete Syntax

∀*expr*∈ <expression>:
  (*expr.parentAS0*∈ <default initialization>∪<timer default initialization>∪<field default initialization> ∪

<internal synonym definition item>∪<open range>∪<transition option>∪
<variables of sort>) ⇒
isConstantExpression$_0$(*expr*)

A <constant expression> must not contain an <active primary>.

∀*expr*∈ <expression>: *expr.parentAS0*∈ <number of instances>∪<named number>∪<transition option>⇒
isSimpleExpression$_0$(*expr*)

A <simple expression> must contain only literals, operators, and methods defined within the package Predefined, as defined in Annex D.

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*>)
<expression gen primary>(<hyphen>,*x*)	=8=> <operator application>(“-”,<*x*>)
<expression gen primary>(**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

Get the one of the possible static sort for an <expression>.

staticSort$_0$(*expr*:<expression>):<sort>=$_{\text{def}}$
take(*expr.staticSortSet*$_0$)

Determine if an <expression> is a <constant expression>. A <constant expression> must not contain an <active primary>.

isConstantExpression$_0$(*expr*: <expression>): *BOOLEAN*=$_{\text{def}}$
¬∃*ap*∈ <active primary>: *isAncestorAS0*(*expr*, *ap*)

(*expr.parentAS0*∈ <default initialization>∪<timer default initialization>∪<field default initialization> ∪
<internal synonym definition item>∪<open range>∪<transition option>∪
<variables of sort>) ∨
isSimpleExpression$_0$(*expr*)

Determine if an <expression> is a <simple expression>. A <simple expression> must contain only literals, operators, and methods defined within the package Predefined, as defined in Annex D.

isSimpleExpression$_0$(*expr*: <expression>): *BOOLEAN*=$_{\text{def}}$
(∀*id*∈ <identifier>: *isAncestorAS0*(*expr*, *id*)∧*id.idKind*$_0$∈ {**literal**, **operator**, **method**}⇒
getEntityDefinition$_0$(*id*, *id.idKind*$_0$)∈ *PREDEFINEDDEFINITION*$_0$)∧
isConstantExpression$_0$(*expr*)

2.9.10.2 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*))

2.9.10.3 Synonym

Concrete Syntax

<synonym> :: <synonym<identifier>

Transformations

<synonym>(*ident*)
provided *ident.refersto*$_0$ ∈ <internal synonym definition item>
=15=>
ident.refersto$_0$.**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>.

2.9.10.4 Extended Primary

Concrete Syntax

<extended primary> =
<indexed primary>
| <field primary>
| <composite primary>
<indexed primary> :: <primary> [<actual parameter>]+
<field primary> :: [<primary>] <field name>
<field name> = <name>
<composite primary> :: <qualifier> [<actual parameter>]+

Transformations

<indexed primary>(*prim*, *params*)
=8=> <method application>(*prim*, <identifier>(<>, "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 12.2.1.

<field primary>(*prim*, *field*)
provided prim ≠ *undefined*
=8=> <method application>(*prim*, *modifyExtractName*(*field*, "Modify"), <>)

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 12.2.1. The transformation according to this model is performed before the modification of the signature of methods in 12.1.4.

<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 12.2.1.

2.9.10.5 Equality Expression

Abstract Syntax

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>.$staticSortSet_0$ **in**
let *set2* = *ee*.**s2-**<expression>.$staticSortSet_0$ **in**
∃*s1*∈*set1*:∃*s2*∈*set2*:$isSortCompatible_0$(*s1*, *s2*)∨ $isSortCompatible_0$(*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=> <expression gen primary>(**not**, <equality expression>(*x*, <equals sign>, *y*))

Mapping to Abstract Syntax

| <equality expression>(*first*, <equals sign>, *second*)
=> **mk-***Equality-expression*(*Mapping*(*first*), *Mapping*(*second*))

2.9.10.6 Conditional Expression

Abstract Syntax

Conditional-expression	::	*Boolean-expression*
		Consequence-expression
		Alternative-expression
Consequence-expression	=	*Expression*
Alternative-expression	=	*Expression*

Conditions on Abstract Syntax

$\forall c \in$ *Conditional-expression*:
(*c*.**s-***Consequence-expression.staticSort*$_1$= *c*.**s-***Alternative-expression.staticSort*$_1$) $\wedge$
(*c*.**s-***Boolean-expression.staticSort*$_1$= **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

$\forall$*ce*$\in$ <conditional expression>:
let *set1*=*ce*.**s-**<consequence expression>*.staticSortSet*$_0$ **in**
let *set2*=*ce*.**s-**<alternative expression>*.staticSortSet*$_0$ **in**
|*set1*| =1 $\wedge$ |*set2*| =1$\wedge$ *isSameSort*$_0$(*set1.take*, *set2.take*)
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*))

2.9.10.7 Operation Application

Abstract Syntax

Operation-application	::	*Operation-identifier* [*Expression*]*

Conditions on Abstract Syntax

∀*oa* ∈ *Operation-application*:
let *os* = *getEntityDefinition*$_1$(*oa*, **operation**) **in**
isActualAndFormalParameterMatched$_1$(*oa*.**s-***Expression*-**seq**, *os*.*formalParameterSortList*$_1$)
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 parameter>]*

<method application> :: <primary> <operation identifier> [<actual parameter>]*

Conditions on Concrete Syntax

∀*methodApp*∈ <method application>:
getEntityDefinition$_0$(*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*.

actualParameterListOfOpId$_1$(*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

2.9.10.8 Range Check Expression

Abstract Syntax

Range-check-expression :: *Range-condition Expression*

Concrete Syntax

<range check expression> ::
<expression> { <range check expression gen identifier> | <sort> }

<range check expression gen identifier> :: <sort><identifier> <constraint>

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**
isSameSort$_0$(*rcExpr*.**s-**<expression>.*staticSort*$_0$, *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>(*,*))
=8=>
if *i.refersto*$_0$ ∈ <syntype definition> **then**
<range check expression>(<range check expression gen identifier>(*i*,
i.refersto$_0$.s-<syntype definition gen syntype>.**s-**<constraint>)
else
<expression gen primary>(*undefined*,
<literal>(<identifier>(< <path item>(**package**, <name>("Predefined"),
<path item>(**type**, <name>("Boolean") >),"True"))
endif

<range check expression>(*sort*)
provided *sort* ∈ <sort> \ <identifier>
=8=>
<expression gen primary>(*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 expression gen identifier>(*, *constraint*) => *Mapping*(*constraint*)

2.9.10.9 Variable Definition

Abstract Syntax

Variable-definition :: *Variable-name*
Sort-reference-identifier
[*Constant-expression*]

Conditions on Abstract Syntax

∀*d* ∈ *Variable-definition*: *d*.**s-***Constant-expression*≠ *undefined* ⇒
d.**s-***Constant-expression*. *staticSort*$_1$= *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> :: [**exported**] <variables of sort>+

<variables of sort> ::
{<variables of sort gen name>}+ <sort> [<constant expression>]

<variables of sort gen name> :: <variable<name> [<remote variable<identifier>]

Conditions on Concrete Syntax

$\forall ea \in$ <variables of sort gen name>:
ea.**s-**<identifier> $\neq$ *undefined* $\Rightarrow$ *ea.parentAS0.parentAS0.isExported*$_0$

<exported as> may only be used for a variable with **exported** in its <variable definition>.

$\forall d \in$ <agent definition> $\cup$<agent type definition> :
$\neg(\exists v1, v2 \in$ <variables of sort gen name>: *v1* $\neq$ *v2* $\wedge$ *v1*.**s-**<identifier> = *v2*.**s-**<identifier> $\wedge$
v1. *parentAS0.parentAS0* $\in$ <variable definition> $\wedge$
v2. *parentAS0.parentAS0* $\in$ <variable definition> $\wedge$
v1.surroundingScopeUnit$_0$ = *d* $\wedge$ *v2.surroundingScopeUnit*$_0$ = *d*)

Two exported variables in an agent cannot mention the same <remote variable identifier>.

Transformations

< <variable definition>(*exp*, < *v* > $^\frown$ *rest*) > **provided** *rest* $\neq$ *empty* =1=>
< <variable definition> (*exp*, < *v* >), <variable definition> (*exp*, *rest*) >

< <variables of sort> (< *v* > $^\frown$ *rest*, *sort*, *expr*) > **provided** *rest* $\neq$ *empty* =1=>
< <variables of sort> (< *v* >, *sort*, *expr*), <variables of sort> (*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>(< <variables of sort gen name>(*name*,*) >, *sort*, *const*)
=> **mk-***Variable-definition*(*Mapping*(*name*), *Mapping*(*sort*), *Mapping*(*const*))

2.9.10.10 Variable Access

Abstract Syntax

Variable-access = *Variable-identifier*

Concrete Syntax

<variable access> :: { <variable<identifier> | **this** }

Conditions on Concrete Syntax

$\forall va \in$ <variable access>: *va*.**s-implicit** = **this** $\Rightarrow$
(*parentAS0ofKind*(*va*, <operation definition>) $\neq$ *undefined* $\wedge$
parentAS0ofKind(*va*, <operation definition>).*kind*$_0$=**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 operation parameter>.*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 12.1.8.

Mapping to Abstract Syntax

| <variable access>(*identifier*) => **mk-***Variable-identifier*(*Mapping*(*identifier*))

2.9.10.11 Assignment and Assignment Attempt

Abstract Syntax

Assignment :: *Variable-identifier Expression*
Assignment-attempt :: *Variable-identifier Expression*

Conditions on Abstract Syntax

$\forall a \in$ *Assignment*: $\exists d \in$ *Variable-definition*:
(*d*=*getEntityDefinition*$_1$(*a*. **s**-*Variable-identifier*,**variable**)) $\wedge$
isCompatibleTo$_1$(*a*.**s**-*Expression.staticSort*$_1$, *d*.**s**-*Sort-reference-identifier*)

In an *Assignment*, the sort of the *Expression* must be sort compatible to the sort of the *Variable-identifier*.

$\forall aa \in$ *Assignment-attempt*: $\exists d \in$ *Variable-definition*:
(*d*=*getEntityDefinition*$_1$(*aa*. **s**-*Variable-identifier*,**variable**)) $\wedge$
isCompatibleTo$_1$(*d*.**s**-*Sort-reference-identifier*, *aa*.**s**-*Expression*. *staticSort*$_1$)

In an *Assignment-attempt*, the sort of the *Variable-identifier* must be sort compatible to the sort of the *Expression*.

Concrete Syntax

<assignment> :: <variable> <expression>
<variable> = <variable<identifier> | <indexed variable> | <field variable>

Mapping to Abstract Syntax

| <assignment>(*var*,*expr*)
=> **if** isAttempt(*var*.staticType,*expr*.staticType) **then**
mk-*Assignment-attempt*(*var*,*expr*)
else
mk-*Assignment*(*var*,*expr*)
endif

2.9.10.12 Extended Variable

Concrete Syntax

<indexed variable> :: <variable> [<actual parameter>]+

<field variable> :: <variable> <field name>

Transformations

<assignment>(<indexed variable>(*var*, *params*), *expr*)
=8=>
<assignment>(*var*, <method application>(*var*, <identifier>(<>, "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 12.2.1. 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 12.2.1. 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*)

2.9.10.13 Default Initialization

Concrete Syntax

<default initialization> ::[<virtuality>] [<constant expression>]

Conditions on Concrete Syntax

$\forall$*def*$\in$ <data type definition>: $\neg\exists$*d1*,*d2*$\in$ <default initialization>:
d1 $\neq$ *d2*$\wedge$*d1.surroundingScopeUnit*$_0$=*def*$\wedge$*d2.surroundingScopeUnit*$_0$=*def*

$\forall$*sd*$\in$ <syntype definition>:
(*sd*.**s-implicit**.**s-**<default initialization>$\neq$*undefined*)$\Rightarrow$
$\neg\exists$*d*$\in$ <default initialization>: *isDefinedIn*$_0$(*d*, *sd.derivedDateType*$_0$)

A <data type definition> or <syntype definition> must contain at most one <default initialization>.

$\forall$*init*$\in$ <default initialization>:
init.**s-**<constant expression> = *undefined* $\Rightarrow$*init.virtuality*$_0\in$ {**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
getEntityDefinition$_0$(*sort*, **sort**). *getDefaultInitialization* $\neq$ *undefined*
=8=>
<variable definition>(*undefined*, < <variables of sort>(
<variables of sort gen name>(*name*,*) >, *sort*,
getEntityDefinition$_0$(*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 initialisation for a data type or syntype.

getDefaultInitialization(*type*: <data type definition> ∪ <syntype definition>): *BOOLEAN* =$_{\text{def}}$
case *type* **in**
| <syntype definition>(*, <syntype definition gen syntype>(*, *init* = <default initialization>(*,*), *)
=> *init*
| <syntype definition>(*, <syntype definition gen syntype>(*parent*, *, *)
=> *parent.getDefaultInitialization*
else
let *init* = *take*({*d* ∈ <default initialization>: *isDefinedIn*$_0$(*d*, *type*)}) **in**
if *init* = *undefined* then
let *parentSort* = *type.derivedDateType*$_0$ **in**
if parentSort = *undefined* **then**
if *type*.**s-**<data type heading>.**s-**<data type kind> = **object** ∨ *type.isPidSort*$_0$ then
<operator application>(<operation identifier>(<>, "Null"), <>)
else
undefined
endif
else
getDefaultInitialization(*parentsort*)
endif
endlet
else
init
endif

2.9.10.14 Now Expression

Abstract Syntax

Now-expression :: ()

Concrete Syntax

<now expression> :: ()

Mapping to Abstract Syntax

| **NOW** => **mk-***Now-expression*()

2.9.10.15 Import Expression

Transformations

The following transformation is supported in the transformations of remote variables.

The import expression has implied syntax for the importing of the result as defined in 10.6 and also has an implied Variable-access of the implied variable for the import in the context where the <import expression> appears.

let *nn* = *newName* **in**
exp = <import expression>(ident, *))
provided *parentAS0ofKind*(exp,<action>) ≠ *undefined*
∧ exp.*parentAS0*.*parentAS0* ∉ <assignment>
=8=>
<variable access>(<identifier>(*empty*, *nn*))
and
let *asg* = <action statement>(*undefined*,<task>(
<assignment>(*nn*, *exp*, *undefined*)) **in**
< *a* = *parentAS0ofKind*(*exp*, <action>) >
=>
< *asg*, *a* >
)
and
(**let** *dcl* = <variable definition>(*undefined*, < <variables of sort>(
<variables of sort gen name>(*nn*, *undefined*), *ident*.*refersto*$_0$.**s-**<sort> , *undefined*)>) **in**
items=*getEntities*(*exp*)
=> < *dcl* > ⌢ *items*
)

let *nn* = *newName* **in**
exp = <import expression>(ident, *))
provided *parentAS0ofKind*(exp,<terminator>) ≠ *undefined*
∧ exp.*parentAS0*.*parentAS0* ∉ <assignment>
=8=>
<variable access>(<identifier>(*empty*, *nn*))
and
let *asg* = <action statement>(*undefined*,<task>(
<assignment>(*nn*, *exp*, *undefined*)) **in**
< *a* = *parentAS0ofKind*(*exp*, <transition gen action statement>) >
=>
<transition gen action statement>(*a*.**s-**<action statement>**-seq** ⌢ < *asg*>,
a.**s-**<terminator statement>)
)
and
(**let** *dcl* = <variable definition>(*undefined*, < <variables of sort>(
<variables of sort gen name>(*nn*, *undefined*), *ident*.*refersto*$_0$.**s-**<sort> , *undefined*)>) **in**
items=*getEntities*(*exp*)
=> < *dcl* > ⌢ *items*
)

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.

2.9.10.16 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

let *nn* = *newName* **in**
exp = <create expression>(body = <create body>(*target*, *params*))
provided *parentAS0ofKind*(exp,<action>) ≠ *undefined*
=8=>
<variable access>(<identifier>(*empty*, *nn*))
and
(**let** *create* = <create request>(*body*, *undefined*) **in**
let *asg* = <action statement>(*undefined*,<task>(
<assignment>(*nn*, <expression gen primary>(<offspring expression>()), *undefined*)) **in**
< *a* = *parentAS0ofKind*(*exp*, <action>) >
=>
< *create*, *asg*, *a* >
)
and
(**let** *dcl* = <variable definition>(*undefined*, <variables of sort>(
<variables of sort gen name>(*nn*, *undefined*), "Pid", *undefined*)) **in**
items=*getEntities*(*exp*)
=> < *dcl* > ^ *items*
)

let *nn* = *newName* **in**
exp = <create expression>(body = <create body>(*target*, *params*))
provided *parentAS0ofKind*(exp,<terminator>) ≠ *undefined*
=8=>
<variable access>(<identifier>(*empty*, *nn*))
and
(**let** *create* = <create request>(*body*, *undefined*) **in**
let *asg* = <action statement>(*undefined*,<task>(
<assignment>(*nn*, <expression gen primary>(<offspring expression>()), *undefined*)) **in**
< *a* = *parentAS0ofKind*(*exp*, <transition gen action statement>) >
=>
<transition gen action statement>(*a*.**s-**<action statement>**-seq** ^ < *create*, *asg*>,
a.**s-**<terminator statement>)
)
and
(**let** *dcl* = <variable definition>(*undefined*, <variables of sort>(
<variables of sort gen name>(*nn*, *undefined*), "Pid", *undefined*)) **in**
items=*getEntities*(*exp*)
=> < *dcl* > ^ *items*
)

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 and 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 11.13.2).

Mapping to Abstract Syntax

| <self expression> => **mk-***Self-expression*()
| <parent expression> => **mk-***Parent-expression*()
| <offspring expression> => **mk-***Offspring-expression*()
| <sender expression> => **mk-***Sender-expression*()

2.9.10.17 Timer Active Expression

Abstract Syntax

Timer-active-expression :: *Timer-identifier Expression**

Conditions on Abstract Syntax

$\forall t \in$ *Timer-active-expression*:
let d = $getEntityDefinition_1$(*t*.**s-***Timer-identifier*, **timer**) **in**
t.**s-***Expression*.*length* = *d*.**s-***Sort-reference-identifier*. *length* $\wedge$
($\forall i \in$ 1.. *t*.**s-***Expression*. *length* :
$isCompatibleTo_1$(*t*.**s-***Expression*[*i*].$staticSort_1$, *d*.**s-***Sort-reference-identifier*[*i*]))
endlet

The sorts of the *Expression* list in the *Timer-active-expression* 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>*

Mapping to Abstract Syntax

| <timer active expression>(*id*,*l*) =>
mk-*Timer-active-expression*(*Mapping*(*id*), *Mapping*(*l*))

2.9.10.18 Any Expression

Abstract Syntax

Any-expression :: *Sort-reference-identifier*

Concrete Syntax

<any expression> :: <sort>

Conditions on Abstract Syntax

$\forall$*exp*$\in$ <any expression>: $isContainingElements_0$(*exp*.**s-**<sort>)

The <sort> must contain elements.

Mapping to Abstract Syntax

| <any expression>() => **mk-***Any-expression*

Auxiliary Functions

The function $isContainingElements_0$ determines if a <sort> contains elements.

isContainingElements$_0$(*s*:<sort>):*BOOLEAN*=$_{def}$
let *d* = *getEntityDefinition*$_0$(*s*, **sort**) **in**
(*d*∈ *PREDEFINEDSORT*$_0$)∨
(∃*cons*∈ <data type constructor>: *cons.surroundingScopeUnit*$_0$ = *d* ∧
(*cons*∈ <structure definition>∪<choice definition>⇒
∀*sort*∈ <sort>:*sort* **in** *cons.fieldSortList*$_0$⇒*isContainingElements*$_0$(*sort*)))∨
(*d.specialization*$_0$≠*undefined*∧
isContainingElements$_0$(*d.specialization*$_0$.**s-**<type expression>.*baseType*$_0$)∧
(∀*acp*∈ <actual context parameter>:
acp **in** *d.actualContextParameterList*$_0$∧*acp.idKind*$_0$ = **sort**⇒ *isContainingElements*$_0$(*acp*))
endlet

2.9.10.19 State Expression

Abstract Syntax

State-expression :: ()

Concrete Syntax

<state expression> :: ()

Mapping to Abstract Syntax

| <state expression>() => **mk-***State-expression*

2.9.10.20 Value Returning Procedure Call

Abstract Syntax

Value-returning-call-node :: [**THIS**] *Procedure-identifier*
[*Expression*]*

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>⇒
getEntityDefinition$_0$(*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* ⇒
getEntityDefinition$_0$(*procId*, **procedure**) = *parentAS0ofKind*(*procCall*, <procedure definitions>)

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.refersto*$_0$, <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.fullQualifier*$_0$^<path item>(*par.entityKind*$_0$, *par.entityName*$_0$), *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* ^
<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.

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*)
=8=>
<value returning procedure call>(<procedure call body>(*nn*, <>))
and // add the new definition
let *defs*=*parentAS0ofKind*(*p*, <agent type definition>).**s-**<agent structure>.**s-**<entity in agent>**-seq in**
defs => *defs* ^
<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 statement>(*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*))

3 Transformation of SDL Shorthands

This clause details the transformation of the SDL constructs, whose dynamic semantics are given after a transformation to the subset of SDL 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.

The following enumeration details the transformation steps to be performed in order.

1) Lexical transformations:
   a) <macro definition>s and <macro call>s (6.2) are identified lexically and <macro call>s are expanded;
   b) <macro definition>s are removed (also in <package definition>s).

   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) Transformation of <choice definition>.
3) <task>s, <task area>s, and <statement list>s are transformed as defined in 11.13.1 and 11.14.
4) Definition references are replaced by <referenced definition>s (see 7.3).
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>.
   – <operation definition>s are transformed into procedures having anonymous names and having the result as <procedure result>. The resulting <procedure definition> is moved to the enclosing scope unit.
6) The package Predefined is included in the <sdl specification>.
7) Transformation of generic system (see clause 13) and external data (see 12.2.3):
   – 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>s 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.
   – <simple expression>s 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 11.13.5);
- Operations involving <infix operation name>s and their operands transformed to the prefix form (see 12.1.4);
- Structure data type (see 12.1.7.2);
- State list (see 11.2);
- Stimulus list (see 11.2.2);
- Field primary (see 12.2.4);
- Structure primary (see 12.2.4);
- <syntype definition>s with object type or value type (see 12.1.9.5)
- Multiple signals in <output body> (see 11.13.4);
- Multiple timers in <set> and <reset> (see 11.15);
- <channel to channel connections>, <channel definitions> by replacing/extending them with gates (see 10.2);
- Default duration value for timer set (see 11.15);
- Initialization of variables of sorts with default initialization (see 12.3.3.2);
- <stimulus> containing <indexed variable>s and <field variable>s are transformed (see 11.2.2);
- Signal list identifiers to a list of signal identifiers (see 10.4);
- Indexed primary (see 12.2.4);
- Field variable (see 12.3.3.1);
- Indexed variable (see 12.3.3.1);
- <return> with <expression> (see 9.5).

9) Insertion of implicit channels as described in 14.4.

10) Insertion of implicit signal lists as described in 10.2.

11) Replacement of context parameters.

12) Full qualifiers are inserted:

    According to the visibility rules and the rules for resolution by context (6.3), 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 11.2 followed by expansion of asterisk exception handler defined in 11.16.1.
    - Multiple appearance of state is merged (see 11.2).
    - Multiple appearance of exception handler is merged (see 11.14).

14) Trailing commas in <stimulus> (see 11.2.2), <create body> (see 11.13.2), <procedure body> (see 11.13.3) and <output body> are inserted.

15) Synonym identifiers are replaced by the expression they denote (see 12.2.3).

16) Implicit declarations for remote procedures and remote variables are generated.

17) Imported and exported values (see 10.6) are transformed. Then remote procedures (see 10.5) are transformed.

APPENDIX I

INDEX

I.1 Functions

actualContextParameterList0, 22, 46, 58, 188; defined at, **47**

actualParameterList0, 20, 23, 24, 27, 28, 30, 31; defined at, **28**

actualParameterListOfOpId1, 8; defined at, **179**

adaptDefinition, 37, 38; defined at, **38**

agentFormalParameterList0, 23, 31, 57, 58, 64, 65; defined at, **57**

agentInheritedFormalParameterList0, 57; defined at, **57**

agentKind1, 8, 41, 72; defined at, **8**

agentLocalFormalParameterList0, 57, 58, 63, 75; defined at, **57**

agentSignatureSortList0, 57, 58; defined at, **57**

agentTypeLocalProcedureDefinitionSet0: defined at, **42**

allSignalsIn, 86, 87; defined at, **87**

allSignalsOut, 86, 87; defined at, **87**

answerPartRangeConditionList0, 40, 128; defined at, **128**

associatedExceptionHandler0, 119, 140; defined at, **140**

asteriskInputListSet0, 103, 106; defined at, **104**

asteriskMatches, 168; defined at, **168**

asteriskSaveListSet0, 103, 106; defined at, **106**

baseType0, 16, 24, 30, 42, 45, 46, 48, 50, 51, 52, 53, 54, 57, 68, 69, 70, 71, 78, 80, 81, 82, 140, 147, 188; defined at, **46**

baseType1, 111, 113; defined at, **112**

baseTypeKind0, 46; defined at, **46**

bigAnd, 134, 135; defined at, **135**

BigSeq, 5, 44, 45, 87, 115, 145; defined at, **5**

bindInBaseType0, 15, 16; defined at, **16**

bindInLocalDefinition0, 15, 16; defined at, **15**

bindInLocalInterface0, 15, 16; defined at, **16**

bindInUsedPackage0, 15, 16; defined at, **16**

calledProcedure0, 24, 50, 123, 134; defined at, **50**

channelDefinitionSet0, 52, 54; defined at, **53**

completeFormalContextParameter0, 47, 55, 56; defined at, **47**

contextOfExp0, 19, 31; defined at, **31**

contextOfIdentifier0, 19; defined at, **19**

contextParameterAtleastDefinition0, 54, 57, 58, 60, 62, 63, 64; defined at, **56**

createNewType0, 45; defined at, **48**

currentTime: defined at, **3**

defaultValue, 157, 158; defined at, **158**

definedEntryNameSet0, 109; defined at, **110**

definedExitNameSet0, 109; defined at, **110**

definedSignalSet0, 54, 125, 126, 145; defined at, **126**

derivedDataType1, 9, 10; defined at, **10**

derivedDateType0, 22, 26, 62, 149, 150, 172, 173, 183, 184; defined at, **62**

destination0, 42, 52, 85; defined at, **85**

direction0, 42, 52, 64, 67, 82; defined at, **85**

doRename, 150; defined at, **150**

empty, 5, 7, 15, 16, 17, 18, 19, 20, 21, 25, 29, 31, 33, 36, 40, 41, 42, 45, 46, 47, 48, 51, 53, 55, 57, 58, 63, 73, 74, 75, 78, 79, 80, 87, 88, 90, 91, 92, 97, 98, 99, 102, 104, 105, 106, 107, 108, 109, 110, 112, 114, 116, 121, 122, 123, 124, 127, 128, 130, 133, 134, 135, 136, 137, 138, 139, 141, 145, 146, 152, 155, 156, 158, 159, 160, 162, 164, 166, 167, 168, 181, 185, 186, 189; defined at, **3**

entityDefinitionSet0, 68; defined at, **68**

entityKind0, 11, 14, 15, 16, 18, 19, 27, 35, 37, 38, 46, 51, 52, 55, 57, 58, 65, 67, 71, 124, 165, 189; defined at, **11**

entityKind1, 3, 7, 8, 9, 14; defined at, **8**

entityName0, 10, 14, 15, 16, 18, 19, 27, 35, 37, 38, 68, 102, 124, 172, 189; defined at, **10**

entityName1, 7; defined at, **7**

entryPointSet1, 111; defined at, **114**

exceptionHandlerNameSet0, 118, 139, 140; defined at, **118**

exceptionHandlerSet0, 118, 139, 140; defined at, **139**

exitPointSet1, 111; defined at, **114**

fieldNameList0, 156, 157, 158; defined at, **158**

fieldSortList0, 26, 156, 158, 188; defined at, **158**

findconnect, 87; defined at, **87**

findContinueLabel, 40, 127, 128, 129; defined at, **128**

formalContextParameterList0, 46, 58, 78; defined at, **46**

formalContextParameterSublist0, 47; defined at, **47**

formalParameterList1, 80, 108, 123; defined at, **80**

formalParameterSortList0, 24, 30; defined at, **31**

formalParameterSortList1, 8, 122, 123, 125, 137, 141, 179; defined at, **80**

fullIdentifier0, 14, 87; defined at, **14**

fullQualifier0, 15, 19, 86, 124, 159, 160, 189; defined at, **15**

fullQualifier1, 7; defined at, **7**

gateDefinitionSet0, 52, 53, 66, 144; defined at, **53**

gatesignalset0, 144, 145; defined at, **145**

getBindingListSet0, 19, 31; defined at, **19**

getCompositeSort0, 25, 26; defined at, **26**

getCreatedAgentDefinition0, 23, 24, 28, 30; defined at, **28**

getCreateExpSort0, 24, 28; defined at, **24**

getDefaultInitialization, 183, 184; defined at, **184**

getDefinitionInBindingList0, 19, 23, 26, 27, 28, 30, 31; defined at, **27**

getEntities, 82, 83, 85, 92, 95, 97, 99, 105, 122, 123, 128, 163, 185, 186; defined at, **95**

getEntityDefinition0, 14, 15, 18, 22, 24, 26, 27, 28, 30, 34, 36, 46, 50, 51, 52, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 67, 78, 81, 82, 85, 88, 89, 96, 103, 124, 125, 143, 145, 146, 149, 163, 165, 172, 173, 176, 179, 183, 188; defined at, **15**

getEntityDefinition1, 7, 8, 9, 10, 44, 103, 112, 113, 120, 121, 122, 123, 125, 137, 141, 179, 182, 187; defined at, **8**

getFieldSort0, 22, 25, 26; defined at, **22**

getImperativeExpSort0, 26; defined at, **26**

getIndexSort0, 22, 25, 29; defined at, **22**

getItemSort0, 22, 26; defined at, **22**

getLabel, 107, 108; defined at, **107**

getOpAppSort0, 26, 27; defined at, **27**

getPredefinedOpParas0, 27, 31; defined at, **31**

getPredefinedOpResult0, 24, 31; defined at, **32**

getPrimarySort0, 25, 26, 28; defined at, **26**

getProcCallBodySort0, 24, 28; defined at, **24**

getRemoteProcCallBodySort0, 24, 28; defined at, **24**

getStartingScopeUnit0, 15, 16; defined at, **16**

getStartingSuInInterface0, 16, 17; defined at, **17**

getStartingSuInUsedPackage0, 16, 17; defined at, **17**

getStaticSort0, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31; defined at, **28**

getSynonymSort0, 26; defined at, **27**

getUnboundFormalContextParameterList0, 46, 47, 55, 56; defined at, **47**

getVarAccessSort0, 26; defined at, **26**

getVariableSort0, 22, 28, 29; defined at, **22**

handled, 93, 95; defined at, **95**

hasImplicitexception, 91, 95; defined at, **95**

head, 5, 16, 17, 20, 21, 29, 31, 35, 45, 47, 48, 54, 93, 97, 107, 110, 115, 122, 126, 128, 135, 139, 146, 152, 155, 167, 168, 182; defined at, **3**

identifier0, 14, 15, 24, 26, 45, 54, 85, 86, 112, 122, 145, 151, 153, 154, 156, 157, 159, 160, 163, 171; defined at, **15**

identifier1, 3, 7, 8, 14, 81, 125; defined at, **7**

idKind0, 7, 14, 27, 28, 54, 85, 88, 89, 103, 104, 125, 163, 176, 188; defined at, **164**

idKind1, 6, 123; defined at, **7**

implicitName, 90, 91, 92, 93, 94, 95, 97, 98, 100; defined at, **95**

implicitSignaturesAdded, 153, 154, 155, 156, 157, 169; defined at, **155**

impliedBases, 144, 145; defined at, **145**

inheritedChannelDefinitionSet0, 53; defined at, **54**

inheritedExceptionHandlerSet0, 139; defined at, **140**

inheritedFormalContextParameterList0, 46; defined at, **46**

inheritedFormalParameterList1, 80; defined at, **80**

inheritedGateDefinitonSet0, 53; defined at, **53**

inheritedOperations, 147, 148, 150; defined at, **150**

inheritedOutputSignalSet0, 42; defined at, **42**

inheritedProcedureFormalParameterList0, 79; defined at, **80**

inheritedStartSet0, 69; defined at, **70**

inheritedStateNodeSet1, 113; defined at, **113**

inheritedStateSet0, 70; defined at, **71**

inoutSignals, 86; defined at, **86**

inputSignalSet1, 111, 112; defined at, **112**

interfaceDefinitionSet0, 66; defined at, **68**

isAbstractType0, 50, 88; defined at, **50**

isActualAndFormalParameterMatched1, 8, 122, 123, 125, 137, 141, 179; defined at, **80**

isAncestorAS0, 16, 19, 31, 38, 40, 68, 103, 104, 105, 106, 107, 110, 114, 119, 129, 135, 151, 152, 163, 171, 173, 175, 176, 188; defined at, **3**

isAncestorAS1, 23, 31, 111, 112, 119, 120, 127; defined at, **3**

isBoundToActualContextPara0, 18; defined at, **18**

isCompatibleOperationAndSignature0, 162, 163, 164; defined at, **164**

isCompatibleTo1, 10, 80, 103, 120, 127, 182, 187; defined at, **10**

isConnected, 82, 83, 86; defined at, **86**

isConsistentKindTo0, 15, 18, 51; defined at, **18**

isConstantExpression0, 175, 176; defined at, **175**

isContainedInConsExp0, 173; defined at, **173**

isContainingElements0, 187, 188; defined at, **188**

isContextParameterCorresponded0, 18, 54, 57, 58, 59, 60, 61, 62, 63, 64, 165; defined at, **58**

isDefinedIn0, 17, 42, 51, 68, 89, 96, 163, 183, 184; defined at, **17**

isDefinedSym0, 171, 172; defined at, **172**

isDirectlySortCompatible0, 148, 149; defined at, **149**

isDirectSubType0, 17, 46, 58, 69; defined at, **46**

isDirectSuperType1, 9, 10, 80; defined at, **9**

isExported0, 78, 89, 96, 181; defined at, **79**

isInOpenRange1, 9; defined at, **9**

isInPartialRegularExpression, 167, 168; defined at, **167**

isInRegularExpr0, 16, 167, 168; defined at, **167**

isMentionedInDefSelection0, 18; defined at, **18**

isObjectSort0, 142, 148, 149; defined at, **149**

isOperationSignatureCompatible0, 71; defined at, **71**

isOptionalField0, 157; defined at, **158**

isParameterizedType0, 45, 46, 55, 65; defined at, **46**

isPathItemMatched0, 37, 38; defined at, **38**

isPidSort0, 143, 148, 149, 184; defined at, **149**

isPredefSort0, 172; defined at, **172**

isPrivate0, 18, 170; defined at, **170**

isPublic0, 17, 170; defined at, **170**

isQualifierMatched0, 16, 17, 18, 38; defined at, **18**

isReachableFrom0, 117, 118, 119; defined at, **119**

isRedefinedType0, 66; defined at, **67**

isRenamedBy0, 18, 27, 28, 148, 150; defined at, **27**

isRenamedBySpec0, 150; defined at, **148**

isRestrictedByInterface0, 146; defined at, **146**

isSameAgentFormalParameterList0, 65; defined at, **67**

isSameConstructorKind0, 147, 148; defined at, **148**

isSameEntityName0, 15, 16; defined at, **16**

isSameGate0, 66; defined at, **67**

isSameInterface0, 66; defined at, **68**

isSameLiteralSignature0, 62; defined at, **62**

isSameNameClassLiteral0, 62; defined at, **168**

isSameNameClassOperation0, 63; defined at, **168**

isSameOperationSignature0, 14, 62; defined at, **63**

isSameProcedureAndSignature0, 89; defined at, **95**

isSameProcedureFormalParameterList0, 66; defined at, **68**

isSameProcedureSignature0, 59; defined at, **59**

isSameRaises0, 59, 63, 80, 95; defined at, **80**

isSameResult0, 59, 63, 95, 164; defined at, **96**

isSameSort0, 25, 29, 30, 31, 40, 57, 58, 59, 60, 61, 63, 64, 67, 68, 72, 95, 96, 148, 149, 164, 171, 173, 178, 180; defined at, **149**

isSameSortList0, 60, 61, 63, 67; defined at, **149**

isSameStateConnectionPoint0, 66; defined at, **68**

isSatisfyAssignmentCondition0, 21; defined at, **21**

isSatisfyCreateCondition0, 23; defined at, **23**

isSatisfyDecisionCondition0, 21; defined at, **22**

isSatisfyEqualityExpCondtion0, 23, 25; defined at, **25**

isSatisfyExpressionCondition0, 21, 23, 24, 25, 27; defined at, **23**

isSatisfyExtendededPrimaryCond0, 25; defined at, **25**

isSatisfyIndexVariableCondition0, 22; defined at, **22**

isSatisfyMethodAppCondition0, 25, 27; defined at, **27**

isSatisfyOpAppCondition0, 25, 27; defined at, **27**

isSatisfyPrimaryCondition0, 23, 25, 27; defined at, **25**

isSatisfyProcedureCallBodyCondition0, 23, 24; defined at, **24**

isSatisfyRangeCheckCondtion0, 23, 25; defined at, **25**

isSatisfyRemoteProcCallBodyCondition0, 23, 24; defined at, **24**

isSatisfyStaticCondition0, 19, 21, 22; defined at, **21**

isSiglistContaining0, 88, 89; defined at, **89**

isSimpleExpression0, 40, 175, 176; defined at, **176**

isSortCompatible0, 22, 23, 24, 25, 26, 27, 71, 148, 149, 177; defined at, **148**

isSubSort0, 149; defined at, **149**

isSubtype0, 46, 51, 52, 55, 57, 58, 60, 62, 63, 64, 66, 68, 126, 147, 150; defined at, **46**

isSuperCounterpart0, 69; defined at, **69**

isSuperType1, 10; defined at, **10**

isTransitionTerminating0, 116, 117; defined at, **117**

isVirtualType0, 65, 66; defined at, **66**

isVisibleIn0, 15, 17, 27; defined at, **17**

isVisibleInDataType0, 17; defined at, **17**

isVisibleInInterface0, 17; defined at, **17**

isVisibleThroughBaseType0, 17, 150; defined at, **17**

isVisibleThroughUseClause0, 18; defined at, **18**

isVisibleThroughUsedPackage0, 17, 18; defined at, **18**

kind0, 11, 69, 164, 165, 181; defined at, **11**

last, 40, 116, 117, 127, 128, 129; defined at, **3**

length, 19, 21, 23, 24, 25, 26, 27, 38, 57, 58, 59, 63, 64, 67, 68, 71, 80, 95, 103, 120, 122, 123, 124, 128, 135, 139, 147, 149, 156, 157, 158, 164, 168, 172, 173, 187; defined at, **3**

localChannelDefinitionSet0, 53; defined at, **53**

localDataTypeDefinitionSet0, 17, 145; defined at, **145**

localExceptionHandlerSet0, 139; defined at, **140**

localFormalContextParameterList0, 45, 46, 88, 143, 146; defined at, **47**

localFormalParameterList1, 80; defined at, **80**

localGateDefinitionSet0, 53, 82; defined at, **53**

localInterfaceDefinitionSet0, 15, 16, 17, 146; defined at, **146**

localOutputSignalSet0, 42; defined at, **42**

localProcedureFormalParameterList0, 79; defined at, **79**

localStartSet0, 69; defined at, **70**

localStateNodeSet1, 113; defined at, **113**

localStateSet0, 70; defined at, **70**

makeProcedureBody, 163; defined at, **166**

Mapping, 5, 14, 33, 36, 43, 44, 45, 46, 49, 53, 65, 74, 79, 84, 85, 88, 101, 102, 104, 105, 106, 107, 108, 110, 114, 115, 116, 118, 119, 120, 121, 123, 124, 126, 128, 130, 131, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 144, 145, 146, 148, 152, 155, 167, 171, 172, 176, 178, 179, 180, 181, 182, 187, 190; defined at, **5**

mergeNumbers, 39; defined at, **39**

mismatchNumber0, 19, 28; defined at, **28**

mismatchNumberInIndexVariable0, 29; defined at, **29**

mismatchNumberOfActualParas0, 30, 31; defined at, **31**

mismatchNumberOfAssignment0, 28; defined at, **28**

mismatchNumberOfCreateExp0, 29; defined at, **30**

mismatchNumberOfDecision0, 28; defined at, **29**

mismatchNumberOfExpression0, 28, 29, 30, 31; defined at, **29**

mismatchNumberOfMethodApp0, 30; defined at, **31**

mismatchNumberOfOpApp0, 30; defined at, **31**

mismatchNumberOfParas0, 30, 31; defined at, **31**

mismatchNumberOfPrimary0, 30, 31; defined at, **30**

mismatchNumberOfProcedureCallBody0, 29; defined at, **30**

mismatchNumberOfRange0, 29; defined at, **29**

mismatchNumberOfRangeCond0, 29; defined at, **29**

mismatchNumberOfRangeCondtionList0, 29; defined at, **29**

mismatchNumberOfRemoteProcCallBody0, 29; defined at, **30**

missingConnections, 83, 85; defined at, **85**

modifyExtractName, 177, 183; defined at, **183**

myImplicitGateIdentifier, 86, 87; defined at, **87**

name0, 10, 33, 41, 69, 73, 82, 89, 93, 94, 95, 147, 148, 150, 162, 163, 172; defined at, **10**

nameList0, 19; defined at, **19**

nameListInActualParameterList0, 20, 21; defined at, **20**

nameListInDecision0, 19, 21; defined at, **21**

nameListInExpression0, 19, 20, 21; defined at, **20**

nameListInExtendedPrimary0, 20; defined at, **20**

nameListInOperationApplication0, 20; defined at, **20**

nameListInPrimary0, 20; defined at, **20**

nameListInRange0, 21; defined at, **21**

nameListInRangeConditions0, 21; defined at, **21**

nameListInRangeList0, 21; defined at, **21**

nameListInVariable0, 19, 21; defined at, **21**

newName, 4, 45, 73, 74, 75, 76, 82, 83, 86, 90, 91, 97, 98, 104, 105, 109, 110, 112, 122, 124, 127, 128, 130, 132, 133, 134, 136, 141, 158, 163, 170, 185, 186, 189; defined at, **4**

nextNumber, 153, 155; defined at, **155**

operationFormalparameterList0, 26, 164; defined at, **164**

operationParameterSortList0, 23, 27, 30, 31; defined at, **27**

operationResult0, 24, 27, 28; defined at, **24**

operationSignatureParameterList0, 27, 63, 71, 164, 172; defined at, **152**

operatorProcedureName, 152, 163; defined at, **152**

origination0, 52, 85; defined at, **85**

parentAS0, 11, 12, 13, 16, 18, 22, 34, 35, 38, 40, 41, 42, 45, 46, 48, 50, 51, 52, 57, 58, 59, 60, 61, 62, 64, 67, 68, 69, 70, 71, 73, 75, 76, 78, 82, 83, 85, 87, 88, 89, 90, 91, 93, 95, 96, 101, 102, 103, 104, 108, 109, 110, 114, 115, 116, 117, 118, 124, 125, 126, 127, 129, 134, 148, 150, 151, 153, 154, 155, 156, 157,

158, 162, 163, 164, 166, 169, 174, 175, 176, 181, 185, 188; defined at, **3**

parentAS0ofKind, 12, 19, 26, 31, 40, 58, 78, 85, 89, 92, 96, 99, 108, 119, 122, 123, 124, 131, 134, 143, 149, 151, 157, 158, 163, 164, 165, 181, 182, 185, 186, 188, 189; defined at, **3**

parentAS1, 3, 7, 9, 22, 23, 26, 31, 41, 72, 81, 101, 111, 125, 127, 141, 179; defined at, **3**

parentAS1ofKind, 7, 111, 117, 125, 138, 140; defined at, **3**

parentDataType0, 62; defined at, **62**

possibleBindingListSet0, 19, 21; defined at, **21**

possibleDefinitionSet0, 21, 27; defined at, **27**

possibleInstances0, 122; defined at, **123**

predefExcDefined, 95, 98; defined at, **95**

procedureFormalParameterList0, 31, 59, 66, 80, 95, 123; defined at, **79**

procedureResult0, 24; defined at, **24**

procedureSignatureParameterList0, 59, 95; defined at, **80**

program: defined at, **3**

qualifier0, 12, 35, 37, 38, 158; defined at, **12**

range1, 9, 127; defined at, **9**

rangeConditionList0, 21, 22, 29, 128; defined at, **128**

rangeOperator, 172; defined at, **173**

reachableExceptionHandlerSet0, 118, 119; defined at, **119**

referencedBy0, 12, 37, 38; defined at, **38**

referencedDefinition0, 15, 16, 37, 38, 65, 70, 102; defined at, **38**

referenceKind0, 11, 12, 38; defined at, **12**

referenceName0, 10, 11, 38; defined at, **11**

refersto0, 14, 45, 55, 56, 82, 86, 87, 88, 90, 91, 92, 93, 94, 97, 98, 100, 104, 115, 122, 123, 124, 137, 157, 158, 176, 180, 185, 189; defined at, **14**

regularElementMatches, 168; defined at, **168**

regularExpressionRepeated, 168; defined at, **168**

replaceContextParameters0, 45, 48; defined at, **48**

replaceInSyntaxTree, 48, 78, 169; *defined at*, **3**

representSameDynamicOpSig0, 19; defined at, **19**

resolutionByContainer0, 15, 16; defined at, **15**

resolutionByContext0, 15, 19, 169; defined at, **19**

rootNodeAS0, 97; defined at, **3**

rootNodeAS1: defined at, **3**

Self: defined at, **3**

signalSet0, 18, 42, 51, 52, 54, 64, 82, 126, 145; defined at, **54**

simpleMapping, 75; defined at, **75**

specialization0, 16, 17, 22, 24, 42, 46, 53, 54, 57, 65, 66, 68, 70, 71, 80, 88, 140, 169, 188; defined at, **65**

specializeArgument, 150; defined at, **150**

startSet0, 48, 69, 70, 109, 110; defined at, **69**

stateConnectionPointSet0, 66; defined at, **68**

stateNameSet0, 102, 103; defined at, **102**

stateNodeSet1, 101, 108, 112, 113; defined at, **113**

stateSet0, 69, 70, 71, 103, 109, 110; defined at, **70**

staticSort0, 40, 89, 96, 125, 132, 171, 180; defined at, **175**

staticSort1, 9, 80, 120, 127, 178, 180, 182, 187; defined at, **9**

staticSortSet0, 27, 31, 173, 175, 177, 178; defined at, **31**

superCounterpart0, 19, 66, 68, 71, 152; defined at, **69**

surroundingScopeUnit0, 12, 14, 15, 16, 17, 18, 26, 27, 28, 35, 38, 42, 46, 47, 50, 51, 52, 53, 57, 58, 62, 65, 66, 67, 68, 69, 78, 82, 86, 101, 102, 103, 105, 109, 111, 117, 118, 122, 125, 128, 139, 140, 145, 146, 147, 169, 171, 172, 181, 183, 188; defined at, **12**

tail, 5, 16, 17, 20, 21, 29, 31, 47, 48, 54, 128, 135, 146, 152, 155, 167, 168; defined at, **3**

take, 8, 12, 15, 16, 17, 19, 22, 26, 27, 38, 40, 47, 57, 65, 66, 67, 68, 69, 70, 80, 82, 83, 87, 140, 150, 157, 165, 169, 175, 178, 180, 184; defined at, **3**

TerminatingDecision, 127, 128, 129; defined at, **129**

TerminatingTransition, 129; defined at, **129**

toSet, 18, 22, 36, 40, 43, 44, 45, 53, 79, 82, 83, 86, 88, 95, 102, 103, 104, 110, 113, 118, 126, 128, 130, 134, 139, 141, 143, 144, 145, 146, 152, 156, 158, 172; defined at, **3**

unconnectedGates, 82, 83, 85, 86; defined at, **85**

updateVirtuality, 152; defined at, **152**

usedEntryNameSet0, 109; defined at, **110**

usedExitNameSet0, 109; defined at, **110**

usedPackage0, 16, 18, 35, 36; defined at, **36**

usedPackageDefinitionList0, 16, 36; defined at, **36**

usedSignalSet0, 54, 125, 126, 145; defined at, **126**

validOutputSignalSet0, 42, 89, 96; defined at, **42**

value0, 40, 62, 73, 75, 153, 168; defined at, **75**

value1, 9, 75; defined at, **9**

virtuality0, 62, 65, 66, 67, 68, 69, 71, 151, 169, 183; defined at, **67**

virtualTypeAtleastDefinition0, 65, 66, 68; defined at, **67**

virtualTypeInheritsDefinition0, 66; defined at, **68**

visibility0, 62, 155, 169, 170; defined at, **155**

I.2 Domains

Agent, 166; defined at, **2**

AgentKind1, 6, 8; defined at, **6**

Binding0, 7; defined at, **7**

BindingList0, 19, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31; defined at, **7**

Boolean, 9, 10, 16, 17, 18, 19, 21, 22, 23, 24, 25, 27, 38, 46, 50, 58, 59, 62, 63, 66, 67, 68, 69, 71, 79, 80, 86, 89, 95, 96, 117, 119, 129, 146, 148, 149, 155, 158, 164, 167, 168, 170, 172, 173, 175, 176, 178, 184, 188; defined at, **2**

Context0, 19, 21, 28, 31; defined at, **7**

EntityDefinition0, 7, 10, 11, 12, 14, 15, 16, 17, 18, 19, 21, 27, 35, 37, 46, 51, 56, 65, 66, 68, 85, 95; defined at, **6**

EntityDefinition1, 3, 5, 7, 8, 9, 10, 14; defined at, **5**

EntityKind0, 11, 12, 15, 16, 18, 46, 164; defined at, **6**

EntityKind1, 6, 7, 8; defined at, **6**

FormalContextParameter0, 6, 11, 18, 54, 55, 56, 58; defined at, **6**

Nat, 19, 28, 29, 30, 31, 72, 75, 105, 166, 168; defined at, **2**

PredefinedDefinition0, 6, 176; defined at, **6**

PredefinedLiteral0: defined at, **6**, **27**

PredefinedOperation0, 6, 24, 27, 31, 32; defined at, **6**

PredefinedSort0, 6, 172, 188; defined at, **6**

Program: defined at, **2**

Real: defined at, **2**

Reference0: defined at, **6**, **10**, **11**, **12**, **15**, **35**, **37**, **38**

ScopeUnit0, 12, 15, 16, 17, 18, 36, 68, 145, 146, 151; defined at, **6**

Signal0, 42, 52, 54, 126, 145; defined at, **7**

Signal1, 112; defined at, **6**

SignalDirection0, 85; defined at, **85**

Symbol0, 172; defined at, **7**

Token, 13, 156, 157, 158, 166, 173, 183; defined at, **2**

TypeDefinition1, 9, 101, 112, 113; defined at, **6**

TypeReference0, 6, 11; defined at, **6**

value1, 9; defined at, **6**

X, 5; defined at, **2**

I.3 AS0 Nonterminals

<abstract>, 41, 50; defined at, **50**

<action statement>, 40, 90, 92, 93, 98, 100, 104, 107, 108, 115, 116, 117, 122, 127, 129, 131, 132, 133, 141, 185, 186; defined at, **115**

<action>, 115, 116, 117, 185, 186; defined at, **115**

<active primary>, 174, 175; defined at, **174**

<actual context parameter>, 18, 45, 47, 54, 56, 57, 58, 59, 60, 61, 62, 63, 64, 165, 188; defined at, **54**

<actual parameter>, 20, 21, 22, 25, 26, 28, 29, 89, 101, 117, 121, 122, 123, 125, 176, 179, 182; defined at, **122**

<agent additional heading>, 33, 41, 42, 43, 55, 57, 72, 75; defined at, **72**

<agent body gen start>, 70, 73; defined at, **73**

<agent body>, 33, 41, 70, 71, 72, 74, 107, 118, 119; defined at, **73**

<agent constraint gen atleast>, 56, 58, 164; defined at, **58**

<agent constraint>, 56, 58; defined at, **58**

<agent context parameter>, 6, 11, 42, 47, 54, 56, 58; defined at, **58**

<agent definition>, 6, 26, 28, 31, 33, 36, 39, 42, 46, 53, 54, 57, 69, 70, 71, 72, 78, 85, 89, 95, 96, 102, 104, 106, 118, 122, 123, 139, 140, 181; defined at, **72**

<agent instantiation>, 39, 74, 76; defined at, **72**

<agent kind>, 11, 57, 58; defined at, **57**

<agent reference>, 6, 72; defined at, **36**

<agent signature>, 57, 58; defined at, **58**

<agent structure>, 33, 42, 43, 44, 70, 71, 73, 75, 76, 95, 124, 144, 189; defined at, **72**

<agent type additional heading>, 42, 43, 47, 48, 55, 74; defined at, **41**

<agent type constraint>, 56, 57; defined at, **57**

<agent type context parameter>, 6, 11, 47, 54, 56, 57, 164; defined at, **57**

<agent type definition>, 6, 26, 28, 31, 34, 36, 39, 42, 53, 54, 57, 69, 70, 71, 73, 78, 82, 83, 85, 86, 89, 95, 96, 102, 118, 122, 123, 124, 125, 139, 140, 181, 189; defined at, **41**

<agent type reference>, 6, 34, 39, 73; defined at, **37**

<algorithm action statement>, 129, 131; defined at, **131**

<algorithm answer part>, 132, 133; defined at, **132**

<algorithm else part>, 132, 133; defined at, **132**

<alternative expression>, 20, 25, 30, 178; defined at, **178**

<alternative question>, 40; defined at, **40**

<alternative statement>, 132, 133; defined at, **132**

<anchored sort>, 142, 143, 148, 149, 150, 152, 153, 154, 157, 165; defined at, **142**, **143**

<answer part>, 13, 40, 122, 127, 128, 129, 133; defined at, **127**

<answer>, 40, 92, 98, 99, 127, 128; defined at, **127**

<any expression>, 26, 128, 165, 174, 187; defined at, **187**

<argument virtuality>, 71, 72, 148, 151, 152, 162; defined at, **151**

<argument>, 71, 148, 150, 151, 152, 153, 154, 156, 159, 166; defined at, **151**

<assignment statement>, 129, 131, 132; defined at, **131**

<assignment>, 7, 19, 21, 28, 31, 92, 93, 94, 98, 100, 104, 121, 131, 132, 141, 165, 182, 183, 185, 186; defined at, **21**, **182**

<asterisk connect list>, 114, 115; defined at, **114**

<asterisk exception handler list>, 118, 139; defined at, **139**

<asterisk exception stimulus list>, 141; defined at, **141**

<asterisk input list>, 69, 103, 104, 106; defined at, **103**

<asterisk save list>, 69, 103, 106; defined at, **106**

<asterisk state list>, 92, 99, 101, 102, 103, 109; defined at, **101**

<asterisk>, 12, 69, 92, 98, 110, 115, 141, 166, 168, 174, 175

<base type>, 22, 45, 46, 48, 52, 54, 82, 88, 123, 147, 148; defined at, **45**

<basic sort>, 142, 143, 148, 149; defined at, **142**

<binary expression>, 20, 23, 28, 30, 174, 175; defined at, **174**

<bit string>, 13; defined at, **13**

<block definition>, 10, 11, 33, 39, 72, 74; defined at, **76**

<block heading>, 10, 39, 74, 76; defined at, **76**

<block reference>, 12, 36, 37; defined at, **37**

<block type definition>, 10, 11, 39, 41, 43, 48, 65, 66, 67, 68, 69, 74, 123, 143; defined at, **43**

<block type heading>, 10, 39, 43, 48, 67, 74, 143; defined at, **43**

<block type reference>, 12, 37, 38, 65, 67, 164; defined at, **37**

<break statement>, 129, 134, 135, 136; defined at, **135**

<call statement>, 129, 131; defined at, **131**

<channel definition>, 6, 39, 42, 53, 54, 72, 75, 76, 82, 83, 84, 86, 87, 91, 97; defined at, **81**

<channel endpoint>, 51, 75, 76, 81, 82, 85, 86, 87, 165; defined at, **81**

<channel path>, 42, 51, 52, 75, 76, 81, 84, 85, 86, 87, 91, 97, 165; defined at, **81**

<channel to channel connection>, 40, 72, 86, 87, 165; defined at, **86**

<character string>, 13, 33, 166, 167, 168; defined at, **13**

<choice definition>, 148, 153, 158, 160, 169, 170, 188, 190; defined at, **158**

<choice of sort>, 158, 161, 170; defined at, **158**

<closed range>, 21, 29, 128, 171, 172; defined at, **171**

<communication constraint>, 89, 96, 125, 165; defined at, **125**

<composite primary>, 20, 25, 26, 176, 177; defined at, **176**

<composite state body>, 41, 44, 69, 70, 71, 107, 108, 109, 110, 118, 119; defined at, **109**

<composite state heading>, 10, 39, 41, 57, 108, 110, 112; defined at, **108**

<composite state reference>, 6, 12, 38, 70, 102, 111; defined at, **37**

<composite state structure>, 41, 44, 45, 69, 70, 71, 108, 110, 112, 115; defined at, **108**

<composite state type constraint>, 56, 63, 64; defined at, **63**

<composite state type context parameter>, 6, 11, 47, 54, 56, 63, 164; defined at, **63**

<composite state type definition>, 6, 10, 11, 31, 34, 36, 39, 44, 45, 48, 53, 57, 65, 66, 67, 68, 69, 70, 71, 73, 77, 101, 102, 109, 110, 112, 117, 118, 139, 140; defined at, **44**

<composite state type heading>, 10, 44, 45, 47, 48, 55, 57, 67, 112; defined at, **44**

<composite state type reference>, 6, 12, 34, 38, 39, 65, 67, 73, 77, 109, 164; defined at, **37**

<composite state type signature>, 63; defined at, **63**

<composite state>, 6, 10, 11, 31, 36, 39, 41, 53, 57, 68, 69, 70, 101, 102, 109, 110, 111, 112, 115, 117, 118, 139, 140; defined at, **108**

<compound statement>, 6, 18, 77, 121, 129, 131, 132, 136, 166; defined at, **130**

<concatenation sign>, 12, 174, 175

<conditional expression>, 20, 25, 26, 30, 75, 174, 178; defined at, **178**

<connect list>, 114, 115; defined at, **114**

<connect part>, 67, 101, 114, 115; defined at, **114**

<consequence expression>, 20, 25, 26, 30, 178; defined at, **178**

<consequence statement>, 132; defined at, **132**

<constant expression>, 23, 27, 40, 137, 156, 158, 171, 172, 173, 174, 175, 176, 181, 183, 184; defined at, **174**

<constant>, 29, 128, 171, 172; defined at, **171**

<constraint>, 170, 171, 172, 180; defined at, **171**

<continuous expression>, 105, 188; defined at, **105**

<continuous signal>, 67, 69, 101, 105, 117, 188; defined at, **105**

<create body>, 122, 123, 131, 165, 186, 191; defined at, **122**

<create expression>, 20, 23, 24, 28, 29, 30, 174, 186, 187; defined at, **186**

<create request>, 115, 122, 123, 186; defined at, **122**

<dash nextstate>, 92, 94, 99, 117, 118; defined at, **117**

<data definition>, 34, 39, 73, 77, 108, 162, 190; defined at, **142**

<data type constructor>, 22, 143, 147, 148, 169, 188; defined at, **153**

<data type definition body>, 22, 143, 145, 147, 148, 153, 156, 158, 159, 160, 170; defined at, **143**

<data type definition>, 6, 10, 11, 14, 17, 22, 26, 48, 62, 142, 143, 144, 145, 147, 148, 149, 150, 151, 152, 153, 154, 155, 157, 158, 159, 160, 161, 163, 164, 169, 170, 183, 184; defined at, **143**

<data type heading>, 10, 11, 47, 48, 143, 144, 149, 158, 159, 160, 170, 184; defined at, **143**

<data type kind>, 37, 61, 143, 149, 184; defined at, **143**

<data type reference>, 6, 12, 34, 39, 73, 77, 109, 164; defined at, **37**

<data type specialization>, 46, 62, 65, 143, 147, 148, 150, 152, 153, 160, 169, 170; defined at, **147**

<decision body>, 40, 122, 127, 128, 129, 133; defined at, **127**

<decision statement body>, 132, 133; defined at, **132**

<decision statement>, 129, 132, 133; defined at, **132**

<decision>, 7, 13, 19, 21, 22, 28, 29, 31, 92, 98, 99, 115, 116, 117, 122, 127, 128, 129, 132; defined at, **21**, **127**

<default initialization>, 67, 143, 170, 174, 176, 183, 184; defined at, **183**

<definition selection>, 18, 34, 35; defined at, **34**

<definition>, 36, 38; defined at, **36**

<destination>, 89, 90, 94, 96, 100, 125, 126, 165; defined at, **125**

<else part>, 40, 127, 128, 129, 133; defined at, **127**

<empty statement>, 129, 132, 133, 134, 135; defined at, **135**

<entity in agent>, 72, 95, 124, 189; defined at, **73**

<entity in composite state>, 108; defined at, **108**

<entity in data type>, 143; defined at, **143**

<entity in interface>, 126, 146; defined at, **146**

<entity in operation>, 162; defined at, **162**

<entity in package>, 34; defined at, **34**

<entity in procedure>, 77; defined at, **77**

<entry point>, 45, 111, 112, 114, 165; defined at, **112**

<equality expression>, 20, 23, 25, 28, 30, 174, 177, 178; defined at, **177**

<equals sign>, 12, 171, 172, 177, 178

<exception constraint>, 63; defined at, **63**

<exception context parameter gen name>, 6, 11, 47, 63; defined at, **63**

<exception context parameter>, 47, 54, 146; defined at, **63**

<exception definition item>, 6, 10, 11, 91, 98, 138; defined at, **138**

<exception definition>, 34, 40, 73, 77, 91, 98, 109, 138, 143, 146, 162; defined at, **138**

<exception handler list>, 118, 138, 139; defined at, **138**, **139**

<exception handler>, 73, 77, 90, 109, 118, 119, 139, 140, 162; defined at, **138**

<exception statement>, 129, 136; defined at, **136**

<exception stimulus list>, 136, 141; defined at, **141**

<exception stimulus>, 90, 136, 141, 165; defined at, **141**

<exit point>, 45, 111, 112, 114, 165; defined at, **112**

<expanded sort>, 142, 149, 165; defined at, **142**

<export body>, 96, 100, 131; defined at, **96**

<export>, 96, 100, 116, 165; defined at, **96**

<exported>, 77, 78, 79, 89; defined at, **77**

<expression gen primary>, 20, 23, 28, 30, 75, 93, 100, 128, 155, 159, 160, 165, 174, 175, 178, 180, 186; defined at, **174**

<expression statement>, 129, 132; defined at, **132**

<expression>, 4, 7, 19, 20, 21, 22, 23, 25, 26, 28, 29, 30, 31, 40, 78, 89, 96, 105, 120, 122, 124, 125, 127, 129, 131, 132, 133, 134, 135, 137, 174, 175, 176, 177, 178, 179, 180, 182, 183, 187, 191; defined at, **19**, **23**, **174**

<extended primary>, 20, 25, 124, 174; defined at, **25**, **176**

<external operation definition>, 162; defined at, **162**

<external procedure definition>, 77, 78, 79; defined at, **78**

<external synonym definition item>, 165, 173; defined at, **173**

<field default initialization>, 156, 157, 158, 174, 176; defined at, **156**

<field name>, 22, 25, 26, 156, 158, 161, 176, 177, 182, 183; defined at, **176**

<field primary>, 20, 25, 26, 165, 176, 177; defined at, **176**

<field variable>, 21, 22, 28, 104, 141, 166, 182, 183, 191; defined at, **182**

<field>, 156, 157, 158, 161, 165; defined at, **156**

<fields of sort>, 22, 156, 159, 161, 170; defined at, **156**

<finalization statement>, 133, 134; defined at, **133**

<formal agent parameter>, 42, 44, 57, 73, 74, 108; defined at, **73**

<formal context parameter>, 18, 41, 42, 44, 47, 55, 56, 77, 87, 88, 143, 146, 165; defined at, **54**

<formal operation parameter>, 159, 160, 162, 164, 165, 182; defined at, **162**

<formal parameter>, 27, 31, 59, 63, 71, 72, 78, 80, 150, 151, 152, 153, 154, 156, 165, 172, 173; defined at, **151**

<formal procedure parameter>, 77, 78, 79, 163; defined at, **78**

<free action>, 73, 77, 107, 108, 109, 162, 190; defined at, **107**, **190**

<gate constraint>, 42, 51, 52, 53, 64, 67, 81, 82, 85, 86, 145, 165; defined at, **51**

<gate context parameter>, 6, 11, 47, 54, 64; defined at, **64**

<gate in definition>, 53, 66, 67, 73, 82, 83, 85, 86, 108, 145; defined at, **51**

<gate>, 47, 51, 52, 64, 67, 81, 82, 86; defined at, **51**, **82**

<greater than or equals sign>, 12, 171, 172, 174, 175

<greater than sign>, 12, 171, 172, 174, 175

<handle statement>, 136; defined at, **136**

<handle>, 67, 90, 117, 118, 136, 138, 141; defined at, **141**

<hex string>, 13; defined at, **13**

<history dash nextstate>, 117, 118; defined at, **117**

<hyphen>, 12, 174, 175

<identifier>, 7, 11, 14, 15, 16, 17, 19, 22, 24, 25, 26, 28, 30, 34, 35, 36, 37, 45, 47, 50, 51, 52, 53, 54, 56, 57, 58, 60, 63, 64, 65, 67, 73, 74, 77, 78, 80, 81, 82, 85, 86, 87, 88, 89, 90, 91, 95, 96, 97, 98, 103, 104, 112, 121, 122, 123, 124, 125, 128, 132, 133, 135, 137, 141, 142, 143, 144, 146, 152, 153, 154, 155, 157, 158, 159, 160, 163, 164, 170, 176, 177, 179, 180, 181, 182, 183, 185, 186, 187, 188, 189, 190; defined at, **14**

<if statement>, 129, 132; defined at, **132**

<imperative expression>, 26, 105, 106, 163, 174; defined at, **174**

<implies sign>, 12, 174, 175

<import expression>, 26, 96, 98, 99, 100, 106, 116, 165, 174, 184, 185; defined at, **96**

<indexed primary>, 20, 25, 26, 165, 176, 177; defined at, **176**

<indexed variable>, 21, 22, 29, 104, 141, 165, 182, 183, 191; defined at, **22**, **182**

<informal text>, 40, 121, 127; defined at, **33**

<initial number>, 39, 73; defined at, **73**

<input list>, 69, 95, 103, 104, 106; defined at, **103**

<input part>, 67, 69, 91, 92, 93, 94, 95, 98, 99, 100, 101, 103, 104, 107, 117, 188; defined at, **103**

<interaction>, 41, 72; defined at, **72**

<interface constraint>, 56, 64, 164; defined at, **64**

<interface context parameter gen name>, 6, 11, 48, 54, 56, 64; defined at, **64**

<interface context parameter>, 48, 54; defined at, **64**

<interface definition>, 6, 10, 11, 16, 17, 39, 48, 52, 64, 66, 67, 68, 89, 96, 126, 142, 143, 144, 146, 149, 150; defined at, **146**

<interface heading>, 10, 47, 48, 143, 144, 146; defined at, **146**

<interface procedure definition>, 146; defined at, **146**

<interface reference>, 6, 12, 34, 38, 67, 73, 165; defined at, **37**

<interface specialization>, 46, 65, 143, 144, 146, 148; defined at, **147**

<interface use list>, 126, 144, 146, 165; defined at, **146**

<interface variable definition>, 146, 165; defined at, **146**

<internal synonym definition item>, 165, 173, 175, 176; defined at, **173**

<join>, 40, 108, 116, 119, 127, 129; defined at, **119**

<label>, 107, 108, 115, 116, 119, 129, 134, 135; defined at, **107**

<labelled statement>, 129, 134, 135; defined at, **135**

<less than or equals sign>, 12, 171, 172, 174, 175

<less than sign>, 12, 171, 172, 174, 175

<literal identifier>, 20, 75, 159, 160, 163, 176; defined at, **176**

<literal list>, 148, 153, 154, 155, 159, 161, 169, 172; defined at, **153**

<literal name>, 11, 20, 27, 62, 147, 148, 153, 155, 161, 176; defined at, **13**

<literal signature>, 6, 10, 11, 17, 18, 61, 62, 148, 153, 155, 164, 170; defined at, **153**

<literal>, 20, 26, 169, 174, 180; defined at, **176**

<local variables of sort>, 129, 130, 131, 132, 165; defined at, **129**

<loop body statement>, 133, 134; defined at, **133**

<loop break statement>, 129, 134; defined at, **134**

<loop clause>, 133, 134, 135; defined at, **133**

<loop continue statement>, 129; defined at, **134**

<loop statement>, 129, 134; defined at, **133**

<loop step>, 133, 134; defined at, **133**

<loop variable definition>, 133, 134, 135, 165; defined at, **133**

<loop variable indication gen identifier>, 133, 134; defined at, **133**

<loop variable indication>, 133, 134, 165; defined at, **133**

<mandatory field>, 156, 157, 158, 159; defined at, **156**

<maximum number>, 39, 73; defined at, **73**

<method application>, 20, 25, 27, 30, 31, 157, 160, 165, 177, 179, 183; defined at, **179**

<method list>, 147, 148, 150, 151, 156, 157, 158, 160, 161, 163; defined at, **151**

<name class literal>, 16, 62, 153, 167, 168; defined at, **166**

<name class operation>, 63, 151, 166, 167, 168, 169; defined at, **166**

<name>, 4, 7, 10, 11, 13, 14, 15, 16, 18, 19, 20, 21, 22, 26, 27, 28, 31, 32, 34, 35, 37, 38, 42, 43, 44, 45, 46, 47, 48, 49, 51, 57, 58, 59, 60, 61, 63, 64, 67, 68, 73, 75, 76, 77, 78, 79, 80, 81, 87, 88, 89, 90, 92, 95, 96, 97, 98, 99, 100, 101, 102, 103, 105, 107, 108, 109, 110, 112, 114, 115, 117, 118, 119, 120, 128, 129, 130, 133, 134, 135, 137, 138, 139, 140, 143, 146, 150, 152, 153, 154, 155, 156, 157, 158, 159, 160, 162, 164, 166, 167, 168, 169, 170, 172, 173, 176, 180, 181, 182, 183; defined at, **13**

<named number>, 11, 62, 153, 155, 175; defined at, **153**

<nextstate body gen name>, 92, 98, 109, 110, 117, 118; defined at, **117**

<nextstate body>, 117; defined at, **117**

<nextstate>, 92, 98, 99, 107, 109, 110, 116, 118, 119; defined at, **117**

<not equals sign>, 12, 171, 172, 177, 178

<now expression>, 26, 174; defined at, **184**

<number of instances>, 37, 38, 39, 49, 72, 74, 123, 175; defined at, **73**

<offspring expression>, 186, 187; defined at, **186**

<on exception>, 42, 73, 77, 89, 93, 96, 100, 101, 103, 105, 107, 109, 114, 118, 119, 120, 121, 122, 123, 125, 127, 137, 138, 139, 140, 141, 162; defined at, **140**

<open range gen greater than or equals sign>, 171, 172; defined at, **171**

<open range>, 21, 29, 171, 175, 176; defined at, **171**

<operation application>, 20, 26, 27, 28, 132; defined at, **179**

<operation body>, 107, 118, 119, 162, 163, 166; defined at, **162**, **163**

<operation definition>, 6, 11, 26, 36, 67, 69, 131, 139, 140, 159, 160, 162, 163, 164, 169, 181, 182; defined at, **162**, **190**

<operation definitions>, 143, 169; defined at, **162**

<operation heading>, 11, 67, 159, 160, 162, 163, 164, 182; defined at, **162**

<operation identifier>, 20, 27, 31, 155, 163, 179, 184; defined at, **162**

<operation kind>, 11, 12, 162; defined at, **162**

<operation name>, 11, 20, 27, 63, 147, 148, 151, 157, 162, 169; defined at, **13**, **162**

<operation preamble>, 67, 151, 152, 153, 154, 155, 156, 157, 159, 160, 162, 169; defined at, **151**

<operation result>, 96, 159, 160, 162, 163, 164, 165; defined at, **162**

<operation signature>, 6, 11, 14, 17, 18, 19, 24, 27, 61, 62, 63, 67, 69, 71, 147, 148, 150, 151, 152, 153, 154, 155, 156, 157, 159, 161, 162, 163, 164, 166, 167, 169, 170, 171, 172; defined at, **151**

<operation signatures>, 24, 143, 147, 148, 150, 151, 152, 153, 154, 156, 157, 158, 160, 163; defined at, **151**

<operations>, 143, 145, 147, 148, 150, 153, 154, 156, 157, 158, 160, 161, 169; defined at, **143**

<operator application>, 20, 25, 27, 30, 31, 75, 92, 98, 99, 132, 135, 137, 155, 157, 159, 160, 165, 169, 174, 175, 177, 179, 184; defined at, **179**

<operator list>, 11, 147, 148, 150, 151, 153, 154, 156, 157, 158, 160, 161, 163, 166; defined at, **151**

<optional field>, 156, 157, 158, 159; defined at, **156**

<or partial regular expression>, 16, 166, 167, 168; defined at, **166**

<output body gen identifier>, 90, 92, 98, 100, 125, 126; defined at, **125**

<output body>, 90, 92, 98, 100, 125, 126, 131, 165, 191; defined at, **125**

<output>, 90, 92, 94, 98, 100, 115, 125, 126; defined at, **125**

<package definition>, 6, 10, 11, 17, 18, 33, 34, 35, 36, 37, 38, 95, 97, 190; defined at, **34**, **36**

<package heading>, 10, 18, 34, 35, 36, 38; defined at, **34**

<package interface>, 18, 34, 35; defined at, **34**

<package reference>, 6, 12, 34, 37; defined at, **36**

<package use clause>, 15, 16, 18, 34, 35, 36, 42, 43, 44, 75, 76, 77, 108, 143, 146, 162, 164, 170; defined at, **34**

<parameter kind>, 59, 63, 68, 71, 78, 95, 124, 151, 152, 162, 163, 164; defined at, **78**

<parameters of sort>, 6, 73, 74, 78, 79, 159, 160, 162, 163, 164, 165, 182; defined at, **73**

<parent expression>, 186, 187; defined at, **186**

<parent sort identifier>, 62, 170, 184; defined at, **170**

<partial regular expression>, 166, 167; defined at, **166**

<path item>, 14, 15, 19, 32, 37, 38, 124, 128, 143, 144, 180, 189; defined at, **32**

<pid expression>, 26, 89, 94, 96, 100, 126, 174; defined at, **185**

<pid sort>, 142, 143, 149; defined at, **142**

<plus sign>, 12, 166, 167, 168, 174, 175

<primary>, 20, 25, 26, 27, 30, 31, 54, 174, 176, 177, 179; defined at, **20**, **174**

<priority input list>, 104, 105; defined at, **105**

<priority input>, 67, 69, 101, 105, 117; defined at, **105**

<priority name>, 105; defined at, **105**

<procedure body>, 70, 71, 77, 91, 98, 101, 106, 107, 118, 119, 124, 163, 166, 189, 191; defined at, **77**

<procedure call body>, 23, 24, 28, 29, 30, 46, 50, 91, 93, 94, 98, 105, 123, 124, 131, 133, 134, 165, 188, 189, 190; defined at, **123**

<procedure call>, 50, 91, 93, 116, 123, 124, 131; defined at, **123**

<procedure constraint>, 56, 58, 59, 78; defined at, **58**

<procedure context parameter>, 6, 11, 47, 54, 56, 58, 164; defined at, **58**

<procedure definition gen compound statement>, 77, 79, 91, 98; defined at, **77**

<procedure definition>, 6, 10, 11, 24, 38, 39, 42, 48, 50, 66, 67, 77, 78, 79, 80, 95, 105, 109, 124, 131, 163, 164, 189, 190; defined at, **77**

<procedure definitions>, 31, 34, 36, 39, 42, 50, 65, 66, 67, 68, 69, 70, 71, 73, 77, 78, 89, 102, 109, 118, 131, 139, 140, 188; defined at, **77**

<procedure heading>, 10, 24, 39, 47, 48, 55, 59, 67, 77, 78, 79, 89, 91, 95, 98, 105, 124, 134, 163, 188, 189; defined at, **77**

<procedure preamble>, 65, 67, 77, 78, 79, 89, 91, 98, 105, 124, 164, 189; defined at, **77**

<procedure reference>, 6, 12, 34, 37, 39, 65, 67, 73, 77, 164; defined at, **37**

<procedure result>, 24, 59, 77, 78, 79, 94, 95, 96, 98, 134, 163, 165, 188, 189; defined at, **78**, **190**

<procedure signature>, 31, 58, 59, 78, 80, 89, 93, 95, 103, 146; defined at, **78**

<process definition>, 10, 11, 33, 39, 72, 74; defined at, **76**

<process heading>, 10, 39, 74, 76; defined at, **76**

<process reference>, 12, 36, 38; defined at, **37**

<process type definition>, 10, 11, 39, 41, 43, 48, 65, 66, 67, 68, 69, 74, 123, 144; defined at, **43**

<process type heading>, 10, 39, 43, 48, 67, 74, 144; defined at, **43**

<process type reference>, 12, 37, 38, 65, 67, 164; defined at, **37**

<provided expression>, 103, 105, 106, 107, 188; defined at, **105**

<qualifier>, 12, 14, 15, 16, 17, 18, 22, 26, 34, 35, 37, 38, 42, 43, 44, 76, 77, 90, 108, 153, 154, 157, 162, 176, 177, 183, 190; defined at, **14**, **37**

<question>, 21, 22, 29, 127; defined at, **127**

<quoted operation name>, 13; defined at, **13**

<raise body>, 90, 91, 92, 94, 98, 99, 103, 121, 131, 165; defined at, **121**

<raise statement>, 129, 131; defined at, **131**

<raise>, 90, 91, 92, 98, 99, 116, 121; defined at, **121**

<raises>, 59, 63, 77, 78, 80, 90, 95, 103, 151, 162, 164; defined at, **78**

<range check expression gen identifier>, 165, 180; defined at, **180**

<range check expression>, 20, 23, 25, 28, 30, 165, 174, 180; defined at, **180**

<range condition>, 21, 29, 40, 127, 128, 132, 133, 171, 172, 190; defined at, **171**

<range>, 21, 29, 171; defined at, **21**, **171**

<reference sort>, 142, 149, 165; defined at, **142**

<referenced definition>, 12, 33, 37, 38, 190; defined at, **36**, **38**

<regular element>, 166, 167, 168; defined at, **166**

<regular expression>, 16, 155, 166, 167, 168; defined at, **166**

<regular interval>, 166, 167; defined at, **166**

<remote procedure call body>, 24, 30, 50, 89, 91, 165, 188, 189; defined at, **89**

<remote procedure call>, 89, 91, 92, 116; defined at, **89**

<remote procedure context parameter>, 6, 11, 47, 54, 59, 146; defined at, **59**

<remote procedure definition>, 11, 31, 34, 39, 59, 73, 89, 90, 93, 95, 103; defined at, **89**

<remote procedure reject>, 94, 103, 104; defined at, **103**

<remote variable context parameter gen name>, 6, 12, 47, 60; defined at, **60**

<remote variable context parameter>, 47, 54, 60, 146, 165; defined at, **60**, **146**

<remote variable definition gen name>, 6, 12, 96, 97; defined at, **96**

<remote variable definition>, 34, 39, 61, 73, 95, 96, 97, 100, 165; defined at, **96**

<rename pair gen literal name>, 147; defined at, **147**

<rename pair gen operation name>, 147, 160; defined at, **147**

<rename pair>, 27, 147, 148; defined at, **147**

<renaming>, 62, 147, 152, 160, 161; defined at, **147**

<reset body>, 131, 137; defined at, **137**

<reset clause>, 137, 138, 165; defined at, **137**

<reset>, 116, 137, 138, 191; defined at, **137**

<result>, 24, 59, 63, 71, 78, 93, 95, 96, 105, 150, 151, 153, 154, 156, 157, 159, 163, 164, 165, 172; defined at, **151**

<return body>, 78, 106, 120, 131, 159, 160, 189; defined at, **120**

<return statement>, 129, 131, 159, 160; defined at, **131**

<return>, 78, 92, 99, 106, 110, 116, 120, 189, 191; defined at, **120**

<save list>, 69, 95, 104, 106; defined at, **106**

<save part>, 67, 69, 92, 94, 95, 98, 101, 106, 165; defined at, **106**

<scope unit kind>, 19, 32, 38; defined at, **32**

<sdl specification>, 33, 35, 190; defined at, **33**

<select definition>, 34, 40, 73, 77, 108, 162, 190; defined at, **39**

<selected entity kind>, 18, 34, 35; defined at, **34**

<self expression>, 185, 187; defined at, **186**

<sender expression>, 93, 100, 186, 187; defined at, **186**

<set body>, 131, 137; defined at, **137**

<set clause>, 137, 138, 165; defined at, **137**

<set>, 115, 137, 191; defined at, **137**

<signal constraint>, 56, 60, 88; defined at, **60**

<signal context parameter gen name>, 6, 11, 47, 54, 56, 59, 60; defined at, **60**

<signal context parameter>, 47, 54, 146, 164; defined at, **59**

<signal definition item>, 6, 10, 11, 47, 48, 55, 56, 87, 88, 90, 97, 126; defined at, **87**

<signal definition>, 34, 39, 73, 88, 90, 97, 126, 146; defined at, **87**

<signal list definition>, 6, 10, 11, 18, 34, 39, 73, 88, 89, 165; defined at, **88**

<signal list item>, 42, 51, 52, 54, 73, 81, 82, 86, 87, 88, 91, 94, 95, 97, 103, 104, 106, 126, 145, 146, 165; defined at, **88**

<signal reference>, 6, 12, 34, 73, 164; defined at, **37**

<signal signature>, 60; defined at, **60**

<simple expression>, 39, 40, 62, 73, 75, 153, 155, 175, 176, 190; defined at, **174**

<size constraint>, 171, 172; defined at, **171**

<solidus>, 12, 174, 175

<sort constraint>, 56, 61, 62, 165; defined at, **61**

<sort context parameter>, 6, 11, 47, 54, 56, 62, 88, 143, 146; defined at, **61**

<sort signature>, 61, 62; defined at, **61**

<sort>, 14, 22, 23, 24, 25, 26, 27, 28, 31, 32, 56, 57, 58, 59, 60, 61, 62, 63, 64, 67, 68, 71, 72, 73, 78, 79, 87, 90, 94, 95, 96, 98, 129, 133, 134, 137, 138, 143, 146, 148, 149, 150, 151, 152, 156, 157, 158, 159, 160, 162, 163, 164, 170, 172, 173, 175, 180, 181, 184, 185, 187, 188, 189; defined at, **14**, **142**

<specialization>, 18, 44, 45, 46, 55, 56, 65, 66, 73, 77, 87, 88, 148, 150; defined at, **65**, **148**

<spelling term>, 20, 26, 169, 174; defined at, **168**

<spontaneous transition>, 67, 69, 101, 107, 117; defined at, **107**

<start>, 42, 48, 67, 69, 70, 73, 77, 79, 91, 98, 101, 106, 109, 110, 117, 118, 162, 189; defined at, **101**

<state aggregation body>, 45, 108; defined at, **111**

<state connection points>, 66, 68, 108, 109, 114, 115; defined at, **114**

<state entry points>, 44, 45, 110, 112, 114; defined at, **114**

<state exit points>, 44, 45, 110, 112, 114, 115; defined at, **114**

<state expression>, 26, 174, 188; defined at, **188**

<state list item>, 101, 102, 103; defined at, **101**

<state list>, 93, 95, 101, 102, 103, 109, 115; defined at, **101**

<state partition connection gen entry>, 45, 111, 112, 114; defined at, **111**

<state partition connection gen exit>, 45, 111, 112, 114; defined at, **111**

<state partition connection>, 111; defined at, **111**

<state partition>, 39, 69, 70, 71, 72, 73, 111, 112; defined at, **111**

<state>, 69, 70, 71, 73, 77, 92, 93, 94, 95, 98, 99, 100, 102, 103, 104, 106, 107, 108, 109, 110, 117, 118; defined at, **101**

<statement list>, 67, 79, 121, 130, 131, 132, 159, 160, 162, 164, 166, 190; defined at, **129**

<statement>, 77, 129, 130, 131, 132, 133, 135, 136, 163, 166; defined at, **129**

<stimulus>, 91, 92, 94, 98, 99, 103, 104, 105, 106, 165, 191; defined at, **103**

<stop>, 116, 119; defined at, **119**

<string name>, 13, 169; defined at, **13**

<structure definition>, 22, 26, 148, 153, 156, 157, 158, 159, 161, 169, 188; defined at, **156**

<synonym constraint>, 61; defined at, **61**

<synonym context parameter gen name>, 6, 11, 54, 61; defined at, **61**

<synonym context parameter>, 47, 54, 143, 165; defined at, **61**

<synonym definition item>, 173; defined at, **173**

<synonym definition>, 6, 11, 142, 143, 173, 176, 190; defined at, **173**

<synonym>, 26, 27, 173, 174, 176; defined at, **176**

<syntype definition gen syntype>, 11, 128, 165, 170, 171, 180, 184; defined at, **170**

<syntype definition gen type preamble>, 170; defined at, **170**

<syntype definition>, 6, 11, 26, 62, 128, 142, 143, 150, 170, 171, 172, 180, 183, 184, 191; defined at, **170**

<syntype>, 142; defined at, **170**

<system definition>, 10, 11, 33, 35, 72, 73, 75, 76, 90; defined at, **75**

<system heading>, 10, 33, 73, 75; defined at, **75**

<system type definition>, 10, 11, 39, 41, 42, 48, 73, 122, 143; defined at, **42**

<system type heading>, 10, 39, 42, 48, 74, 143; defined at, **42**

<system type reference>, 12, 37, 38, 164; defined at, **37**

<task>, 13, 92, 93, 94, 98, 100, 104, 115, 121, 131, 141, 185, 186; defined at, **121**, **190**

<terminating statement>, 129, 130, 133, 134, 135; defined at, **129**

<terminator statement>, 40, 90, 91, 92, 93, 94, 98, 99, 100, 106, 107, 108, 109, 110, 115, 116, 117, 127, 128, 129, 131, 185, 186, 189; defined at, **116**

<terminator>, 116, 129, 185, 186; defined at, **116**

<textual endpoint constraint>, 51, 52, 53, 67, 164; defined at, **51**

<textual gate definition>, 6, 10, 11, 42, 51, 52, 53, 67, 75, 76, 81, 82, 86, 145; defined at, **51**

<textual interface gate definition>, 51, 52, 53, 67, 145, 164; defined at, **51**

<textual operation reference>, 6, 11, 12, 143, 162; defined at, **162**

<textual system specification>, 33, 97; defined at, **33**

<textual typebased agent definition>, 6, 33, 48, 50, 51, 69, 70, 72, 81, 82, 83, 84, 85; defined at, **48**

<textual typebased block definition>, 10, 11, 33, 48, 49, 52, 74, 123, 144; defined at, **49**

<textual typebased process definition>, 10, 11, 33, 48, 49, 52, 74, 123, 144; defined at, **49**

<textual typebased state partition definition>, 6, 10, 11, 45, 50, 51, 69, 70, 82, 111; defined at, **49**

<textual typebased system definition>, 10, 11, 48, 49, 73, 122, 144; defined at, **49**

<timer active expression>, 165, 174, 187; defined at, **187**

<timer constraint>, 61; defined at, **61**

<timer context parameter gen name>, 6, 11, 47, 61; defined at, **61**

<timer context parameter>, 42, 47, 54; defined at, **61**

<timer default initialization>, 137, 174, 176; defined at, **137**

<timer definition item>, 6, 10, 11, 91, 137; defined at, **137**

<timer definition>, 39, 73, 91, 95, 137; defined at, **137**

<transition gen action statement>, 40, 90, 91, 98, 104, 108, 109, 110, 115, 116, 122, 127, 128, 141, 185, 186; defined at, **115**

<transition option>, 13, 40, 116, 117, 175, 176, 190; defined at, **40**, **116**

<transition>, 40, 92, 93, 94, 98, 99, 100, 101, 103, 104, 105, 107, 114, 116, 117, 118, 119, 127, 128, 129, 131, 141; defined at, **115**

<try statement>, 136; defined at, **136**

<type expression>, 16, 24, 30, 42, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 65, 68, 69, 70, 71, 73, 74, 78, 80, 81, 82, 88, 112, 115, 123, 140, 143, 144, 145, 147, 160, 165, 188; defined at, **45**

<type preamble>, 37, 43, 67, 77, 87, 143, 170; defined at, **41**

<typebased block heading>, 46, 49, 74, 123, 144; defined at, **49**

<typebased composite state>, 46, 50, 51, 69, 70, 101, 102, 103, 112; defined at, **49**

<typebased process heading>, 46, 49, 74, 123, 144; defined at, **49**

<typebased state partition heading>, 45, 46, 49; defined at, **49**

<typebased system heading>, 46, 49, 73, 123, 144; defined at, **49**

<valid input signal set>, 72, 108, 165; defined at, **73**

<value returning procedure call>, 20, 23, 28, 29, 50, 93, 94, 98, 105, 174, 188, 189, 190; defined at, **188**

<variable access>, 26, 124, 159, 160, 174, 181, 182, 185, 186; defined at, **181**

<variable constraint>, 60; defined at, **60**

<variable context parameter gen name>, 6, 12, 47, 60; defined at, **60**

<variable context parameter>, 42, 47, 54, 60, 165; defined at, **60**

<variable definition statement>, 129, 130, 131, 132; defined at, **129**

<variable definition>, 39, 73, 77, 78, 79, 91, 97, 98, 108, 109, 131, 132, 162, 181, 183, 184, 185, 186; defined at, **78**, **109**, **181**

<variable>, 19, 21, 22, 28, 29, 103, 104, 141, 182, 183; defined at, **21**, **182**

<variables of sort gen name>, 78, 79, 91, 96, 98, 165, 181, 183, 185, 186; defined at, **181**

<variables of sort>, 6, 12, 79, 91, 97, 98, 165, 175, 176, 181, 183, 185, 186; defined at, **181**

<via path>, 125, 126, 165; defined at, **125**

<virtuality constraint>, 41, 44, 65, 66, 67, 68, 77, 87, 143, 146, 164; defined at, **65**

<virtuality>, 37, 41, 44, 65, 67, 69, 77, 79, 101, 103, 105, 106, 107, 114, 141, 146, 151, 152, 164, 183; defined at, **65**

<visibility>, 151, 153, 155, 156, 158, 159, 169, 170; defined at, **169**

I.4 AS1 Nonterminals

Action-return-node, 120; defined at, **120**

Active-expression, 173; defined at, **174**

Agent-definition, 5, 7, 8, 33, 41, 43, 44, 49, 72, 81, 112, 113, 121, 122; defined at, **72**

Agent-formal-parameter, 41, 80, 108, 122; defined at, **72**

Agent-identifier, 121, 122, 124; defined at, **13**

Agent-kind, 8, 41, 58, 72; defined at, **41**

Agent-name, 8, 32, 72; defined at, **13**

Agent-qualifier, 32; defined at, **32**

Agent-type-definition, 5, 6, 7, 8, 34, 36, 41, 43, 44, 72, 80, 81, 113, 122, 125; defined at, **41**

Agent-type-identifier, 8, 41, 72, 113, 122; defined at, **13**

Agent-type-name, 8, 32, 41; defined at, **13**

Agent-type-qualifier, 32; defined at, **32**

Alternative-expression, 178; defined at, **178**

Any-expression, 7, 9, 174, 187; defined at, **187**

Argument, 80, 150; defined at, **150**

Assignment, 7, 121, 134, 135, 182; defined at, **7**, **121**, **182**

Assignment-attempt, 7, 121, 182; defined at, **182**

Boolean-expression, 105, 178; defined at, **105**

Break-node, 115, 130, 133, 134, 135; defined at, **130**

Call-node, 7, 112, 115, 123, 124; defined at, **123**

Channel-definition, 5, 8, 41, 43, 44, 81, 84, 125; defined at, **81**

Channel-identifier, 124, 125; defined at, **14**

Channel-name, 8, 81; defined at, **13**

Channel-path, 7, 81, 85, 125; defined at, **81**

Closed-range, 9, 171, 172; defined at, **171**

Composite-state-formal-parameter, 44, 80; defined at, **108**

Composite-state-graph, 44, 110, 113, 120; defined at, **109**

Composite-state-type-definition, 5, 6, 7, 8, 34, 36, 41, 43, 44, 45, 77, 79, 80, 111, 113, 114; defined at, **44**

Composite-state-type-identifier, 44, 72, 101, 110, 113, 117; defined at, **14**

Compound-node, 115, 130, 133, 134, 135, 136; defined at, **130**

Conditional-expression, 9, 75, 173, 174, 178; defined at, **178**

Condition-item, 9, 171; defined at, **171**

Connection-definition, 110, 111; defined at, **110**

Connect-node, 101, 115; defined at, **114**

Connector-name, 107, 119, 130; defined at, **13**

Consequence-expression, 9, 178; defined at, **178**

Constant-expression, 9, 75, 127, 171, 173, 179, 180; defined at, **173**

Continue-node, 115, 134, 135; defined at, **130**

Continuous-expression, 105; defined at, **105**

Continuous-signal, 101, 102, 105; defined at, **105**

Create-request-node, 7, 115, 121, 123; defined at, **121**

Dash-nextstate, 117, 118; defined at, **117**

Data-type-definition, 5, 6, 7, 8, 10, 33, 36, 41, 43, 44, 45, 77, 79, 142, 143, 144, 145; defined at, **142**

Data-type-identifier, 9, 142, 145; defined at, **14**

Data-type-name, 32; defined at, **13**

Data-type-qualifier, 32; defined at, **32**

Decision-answer, 126, 127, 128, 133, 134; defined at, **126**

Decision-node, 115, 127, 128, 133, 134; defined at, **126**

Decision-question, 126, 127; defined at, **126**

DefinitionAS0, 4, 5, 12, 48, 67, 69, 86, 87, 95, 128, 129; *defined at*, **2**

DefinitionAS1, 5; defined at, **2**

Destination-gate, 7, 81; defined at, **81**

Direct-via, 7, 124, 125; defined at, **124**

Dynamic-operation-signature, 142, 143, 144, 145; defined at, **150**

Else-answer, 126, 128, 133; defined at, **126**

Else-handle-node, 138, 139, 141; defined at, **140**

Entry-connection-definition, 110, 114; defined at, **111**

Entry-procedure-definition, 108, 109, 110; defined at, **108**

Equality-expression, 9, 173, 174, 178; defined at, **177**

Exception-definition, 5, 7, 8, 34, 36, 41, 43, 44, 45, 80, 120, 138, 141, 142, 143, 144, 145, 146; defined at, **138**

Exception-handler-name, 138, 140; defined at, **13**

Exception-handler-node, 72, 77, 110, 130, 136, 138, 139, 140; defined at, **138**

Exception-identifier, 120, 140, 141; defined at, **14**

Exception-name, 8, 138; defined at, **13**

Exit-connection-definition, 110, 114; defined at, **111**

Exit-procedure-definition, 108, 109, 110; defined at, **108**

Expanded-sort, 14, 142; defined at, **14**

Expression, 9, 80, 105, 117, 120, 121, 122, 123, 124, 125, 126, 127, 136, 137, 166, 167, 177, 178, 179, 182, 187, 188; defined at, **173**

First-operand, 177; defined at, **177**

Formal-argument, 7, 80, 150, 179; defined at, **150**

Free-action, 72, 77, 108, 110; defined at, **107**

Gate-definition, 5, 7, 8, 41, 43, 44, 53, 81, 125; defined at, **50**

Gate-identifier, 81, 124; defined at, **14**

Gate-name, 8, 50; defined at, **13**

Graph-node, 115, 121, 123, 124, 126, 130, 134, 135, 136, 137, 138; defined at, **115**

Handle-node, 7, 136, 138, 139, 141; defined at, **140**

Identifier, 6, 7, 8, 9, 13, 14, 123, 150, 173, 178; defined at, **13**

Imperative-expression, 174; defined at, **174**

Informal-text, 13, 121, 126; defined at, **33**

Init-graph-node, 130; defined at, **130**

Initial-number, 72; defined at, **72**

Inner-entry-point, 111, 114; defined at, **111**

Inner-exit-point, 111, 114; defined at, **111**

Inout-parameter, 77, 79, 123; defined at, **77**

In-parameter, 77, 79, 123; defined at, **77**

Input-node, 7, 101, 102, 103, 104, 105, 112; defined at, **103**

In-signal-identifier, 50; defined at, **50**

Interface-definition, 8, 9, 142, 146; defined at, **145**

Interface-name, 32; defined at, **13**

Interface-qualifier, 32; defined at, **32**

Join-node, 115, 119; defined at, **119**

Literal, 7, 9, 173; defined at, **6**, **8**, **9**, **13**, **14**, **75**, **142**, **143**, **144**, **145**, **153**, **155**, **176**

Maximum-number, 72; defined at, **72**

Name, 7, 9, 13, 108, 110, 114, 145; defined at, **13**

Nameclass-literal, 153, 166, 167; defined at, **166**

Named-nextstate, 117, 118; defined at, **117**

Named-return-node, 120; defined at, **120**

Named-start-node, 109, 110; defined at, **108**

Nextstate-node, 115, 117; defined at, **117**

Nextstate-parameters, 117, 118; defined at, **117**

Nonvirtual-argument, 150, 152; defined at, **150**

Now-expression, 9, 174, 184; defined at, **184**

Number-of-instances, 49, 72, 74; defined at, **72**

Object-data-type-definition, 8, 142; defined at, **142**

Offspring-expression, 185, 187; defined at, **185**

On-exception, 72, 77, 100, 101, 103, 105, 107, 108, 114, 115, 126, 136, 138, 140; defined at, **140**

Open-range, 7, 9, 171, 172, 179; defined at, **171**

Operation-application, 7, 9, 75, 174, 179; defined at, **178**

Operation-identifier, 9, 171, 178, 179; defined at, **14**

Operation-name, 8, 150; defined at, **13**

Operation-signature, 6, 8, 71, 80, 150, 179; defined at, **150**

Originating-gate, 7, 81, 125; defined at, **81**

Outer-entry-point, 111, 114; defined at, **111**

Outer-exit-point, 111, 114; defined at, **111**

Out-parameter, 77, 79, 123; defined at, **77**

Output-node, 7, 115, 125, 126; defined at, **124**

Out-signal-identifier, 50, 125; defined at, **50**

Package-definition, 5, 8, 33, 36; defined at, **33**

Package-name, 8, 32, 33; defined at, **13**

Package-qualifier, 32; defined at, **32**

Parameter, 7, 72, 74, 77, 80; defined at, **72**

Parent-expression, 185, 187; defined at, **185**

Parent-sort-identifier, 10, 170; defined at, **170**

Partial-regular-expression, 166, 167; defined at, **166**

Path-item, 13; defined at, **32**

Pid-expression, 9, 174; defined at, **185**

Priority-name, 105; defined at, **105**

Procedure-definition, 5, 6, 7, 8, 34, 36, 41, 43, 44, 45, 77, 79, 80, 108, 110, 113, 120, 123; defined at, **76**

Procedure-formal-parameter, 76, 80, 123; defined at, **77**

Procedure-graph, 77, 101, 113, 117, 119, 120, 138, 140; defined at, **77**

Procedure-identifier, 9, 76, 112, 113, 123, 188; defined at, **13**

Procedure-name, 8, 32, 45, 76, 108, 110; defined at, **13**

Procedure-qualifier, 32; defined at, **32**

Procedure-start-node, 77, 101; defined at, **100**

Provided-expression, 103, 107; defined at, **105**

Qualifier, 7, 9, 13; defined at, **13**

Raise-node, 7, 115, 120, 121; defined at, **120**

Range-check-expression, 9, 174, 180; defined at, **179**

Range-condition, 9, 126, 127, 170, 171, 172, 179; defined at, **171**

Reference-sort, 14, 142; defined at, **14**

Regular-Expression, 166, 167; defined at, **166**

Regular-interval, 166, 167; defined at, **166**

Regular-multiplicity, 166; defined at, **166**

Reset-node, 7, 115, 137, 138; defined at, **136**

Result, 7, 9, 71, 76, 120, 150, 153; defined at, **77**

Return-node, 115, 120; defined at, **120**

Save-signalset, 7, 101, 102, 112; defined at, **106**

SDL-specification, 33; defined at, **33**

Second-operand, 177; defined at, **177**

Self-expression, 185, 187; defined at, **185**

Sender-expression, 185, 187; defined at, **185**

Set-node, 7, 115, 137, 138; defined at, **136**

Signal-definition, 5, 7, 8, 34, 36, 41, 43, 44, 80, 88, 103, 125, 145, 146; defined at, **87**

Signal-destination, 7, 124; defined at, **124**

Signal-identifier, 50, 81, 101, 103, 106, 112, 124, 125; defined at, **14**

Signal-name, 8, 32, 87; defined at, **13**

Signal-qualifier, 32; defined at, **32**

Single-literal, 153; defined at, **153**

Size-range, 171, 172; defined at, **171**

Sort, 8, 9, 80, 125, 142, 145; defined at, **9**, **10**, **14**, **72**, **77**, **80**, **87**, **103**, **120**, **123**, **125**, **136**, **137**, **138**, **141**, **145**, **150**, **170**, **180**, **182**, **187**

Spontaneous-transition, 101, 102, 107; defined at, **107**

State-aggregation-node, 44, 45, 111, 113; defined at, **110**

State-entry-point-definition, 44, 111, 114; defined at, **108**

State-entry-point-name, 100, 108, 111, 117; defined at, **13**

State-exit-point-definition, 44, 111, 114; defined at, **108**

State-exit-point-name, 111, 114, 120; defined at, **13**

State-expression, 188; defined at, **188**

State-machine-definition, 7, 41, 44, 112, 113; defined at, **72**

State-name, 8, 32, 72, 101, 117; defined at, **13**

State-node, 5, 7, 8, 72, 77, 101, 102, 110, 112, 113; defined at, **101**

State-partition, 7, 45, 110, 111, 112, 113; defined at, **110**

State-qualifier, 32; defined at, **32**

State-start-node, 72, 101, 110; defined at, **100**

State-transition-graph, 101, 109, 110, 113, 117, 138, 140; defined at, **72**

State-type-name, 8, 32, 44; defined at, **13**

State-type-qualifier, 32; defined at, **32**

Static-operation-signature, 142, 143, 144, 145, 152; defined at, **150**

Step-graph-node, 130; defined at, **130**

Stop-node, 115, 119; defined at, **119**

Syntype-definition, 6, 8, 9, 10, 34, 36, 41, 43, 44, 45, 77, 79, 142, 143, 144, 145, 171; defined at, **170**

Syntype-identifier, 14; defined at, **14**

Syntype-name, 8, 170; defined at, **13**

Task-node, 115; defined at, **121**

Terminator, 115, 118, 119, 120, 121; defined at, **115**

Time-expression, 136; defined at, **136**

Timer-active-expression, 7, 9, 174, 187; defined at, **187**

Timer-definition, 5, 7, 8, 41, 43, 44, 80, 137; defined at, **136**

Timer-identifier, 136, 137, 187; defined at, **14**

Timer-name, 8, 136, 137, 187; defined at, **13**

Transition, 100, 103, 105, 107, 108, 114, 116, 126, 130, 133, 134, 135, 140; defined at, **115**

Value-data-type-definition, 8, 142, 144; defined at, **142**

Value-returning-call-node, 7, 9, 123, 174, 190; defined at, **188**

Value-return-node, 120; defined at, **120**

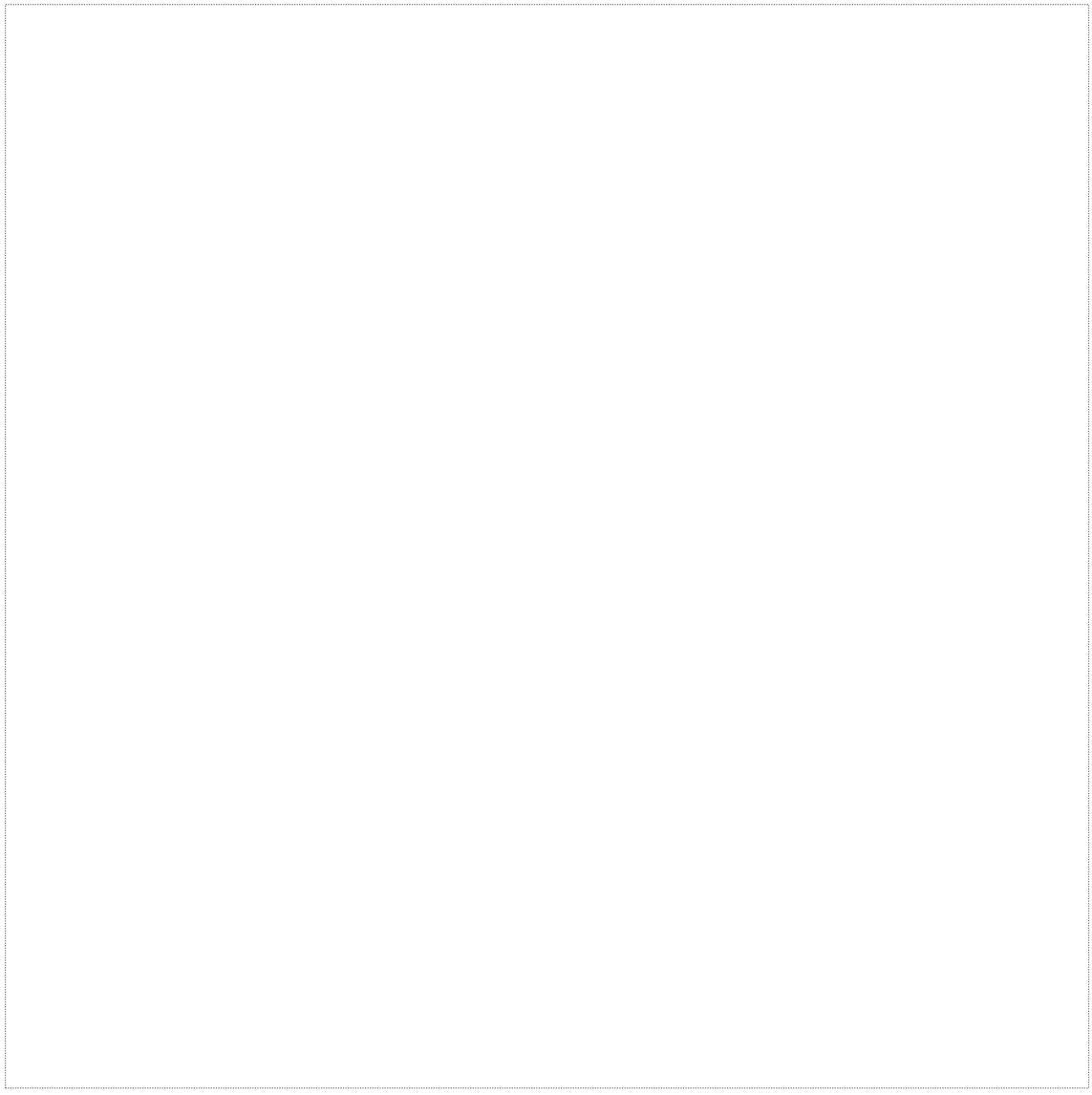
Variable-access, 7, 174; defined at, **181**

Variable-definition, 5, 7, 8, 41, 43, 44, 45, 77, 79, 103, 130, 134, 180, 181, 182; defined at, **180**

Variable-identifier, 7, 9, 103, 123, 140, 141, 181, 182; defined at, **14**

Variable-name, 8, 72, 180; defined at, **13**

Virtual-argument, 150, 152; defined at, **150**



SERIES OF ITU-T RECOMMENDATIONS

Series A	Organization of the work of ITU-T
Series B	Means of expression: definitions, symbols, classification
Series C	General telecommunication statistics
Series D	General tariff principles
Series E	Overall network operation, telephone service, service operation and human factors
Series F	Non-telephone telecommunication services
Series G	Transmission systems and media, digital systems and networks
Series H	Audiovisual and multimedia systems
Series I	Integrated services digital network
Series J	Cable networks and transmission of television, sound programme and other multimedia signals
Series K	Protection against interference
Series L	Construction, installation and protection of cables and other elements of outside plant
Series M	TMN and network maintenance: international transmission systems, telephone circuits, telegraphy, facsimile and leased circuits
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Telephone transmission quality, telephone installations, local line networks
Series Q	Switching and signalling
Series R	Telegraph transmission
Series S	Telegraph services terminal equipment
Series T	Terminals for telematic services
Series U	Telegraph switching
Series V	Data communication over the telephone network
Series X	Data networks and open system communications
Series Y	Global information infrastructure and Internet protocol aspects
Series Z	**Languages and general software aspects for telecommunication systems**

Geneva, 2002