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
<table>
<thead>
<tr>
<th>Formal Description Techniques (FDT)</th>
<th>Z.100–Z.109</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification and Description Language (SDL)</td>
<td>Z.100–Z.109</td>
</tr>
<tr>
<td>Application of Formal Description Techniques</td>
<td>Z.110–Z.119</td>
</tr>
<tr>
<td>Message Sequence Chart</td>
<td>Z.120–Z.129</td>
</tr>
<tr>
<td>Programming Languages</td>
<td></td>
</tr>
<tr>
<td>CHILL: The ITU-T programming language</td>
<td>Z.200–Z.209</td>
</tr>
<tr>
<td>Man-Machine Language</td>
<td></td>
</tr>
<tr>
<td>General principles</td>
<td>Z.300–Z.309</td>
</tr>
<tr>
<td>Basic syntax and dialogue procedures</td>
<td>Z.310–Z.319</td>
</tr>
<tr>
<td>Extended MML for visual display terminals</td>
<td>Z.320–Z.329</td>
</tr>
<tr>
<td>Specification of the man-machine interface</td>
<td>Z.330–Z.399</td>
</tr>
<tr>
<td>Quality of Telecommunication Software</td>
<td>Z.400–Z.499</td>
</tr>
<tr>
<td>Methods for Validation and Testing</td>
<td>Z.500–Z.599</td>
</tr>
</tbody>
</table>

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

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

- 2.6.1 System ................................................................. 75
- 2.6.2 Block ................................................................. 76
- 2.6.3 Process ............................................................... 76
- 2.6.4 Procedure .......................................................... 76

## 2.7 Communication

- 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

- 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

- 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
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](image)

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, =, ≠, ∧, ∨, ⇒, ⇔, ¬, ∃, ∀, >, ≥, <, ≤, |, /

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 nparentAS1 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:

∀d, d' ∈ ENTITYDEFINITION1; d.entityKind1 = d'.entityKind1 ∧ d ≠ d' ⇒ d.identifier1 ≠ d'.identifier1

where:

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

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

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

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.

```plaintext
<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

```plaintext
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:

```plaintext
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:

\[ \text{Mapping} : \text{DefinitionAS0} \rightarrow \text{DefinitionAS1} \]

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.

\[ \text{Mapping}(a: \text{DefinitionAS0}) : \text{DefinitionAS1} = \text{def} \]
\[ \text{case} \ a \ \text{of} \]

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.

\[ \text{BigSeq}(s: X\text{-seq-seq}) : X\text{-seq} = \text{def} \]
\[ \text{if} \ s = \text{empty} \ \text{then} \ \text{empty} \ \text{else} \ s.\text{head} \ \cap \ \text{BigSeq}(s.\text{tail}) \ \text{endif} \]

2.1.6.2 Domain Definitions for AS1

\( \text{SCOPEUNIT}_1 := \) the union of all the scope unit in AS1.

\( \text{SCOPEUNIT}_1 = \text{def} \)
\( \text{Package-definition} \)
\( \cup \)
\( \text{Agent-definition} \)
\( \cup \)
\( \text{Agent-type-definition} \)
\( \cup \)
\( \text{Procedure-definition} \)
\( \cup \)
\( \text{Signal-definition} \)
\( \cup \)
\( \text{Composite-state-type-definition} \)
\( \cup \)
\( \text{Data-type-definition} \)
\( \cup \)
\( \text{State-node} \)

\( \text{ENTITYDEFINITION}_1 := \) the union of all the entity definitions in AS1.

\( \text{ENTITYDEFINITION}_1 = \text{def} \)
\( \text{Package-definition} \)
\( \cup \)
\( \text{Agent-definition} \)
\( \cup \)
\( \text{Agent-type-definition} \)
\( \cup \)
\( \text{Procedure-definition} \)
\( \cup \)
\( \text{Channel-definition} \)
\( \cup \)
\( \text{Gate-definition} \)
\( \cup \)
\( \text{Signal-definition} \)
\( \cup \)
\( \text{Timer-definition} \)
\( \cup \)
\( \text{Exception-definition} \)
\( \cup \)
\( \text{Variable-definition} \)
\( \cup \)
\( \text{Data-type-definition} \)
\( \cup \)
\( \text{State-node} \)
ENTITYKIND: the set of all the entity kinds in AS1.

ENTITYKIND = \{ agent, agent type, package, state, state type, procedure, variable, signal, timer, channel, gate, sort, exception, literal, operation \}

AGENTKIND: the set of agent kinds in AS1.

AGENTKIND = \{ system, block, process \}

SIGNAL = \{ id \in Identifier: id.idKind = \{ signal, timer \} \}

TYPEDEFINITION = \{ Agent-type-definition, Procedure-definition, Composite-state-type-definition, Data-type-definition \}

VALUE = \{ Literal-signature \}

Domain Definitions for AS0

ENTITYKIND = \{ 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 = \{ <agent type definition>, <composite state type definition>, <data type definition>, <procedure definition>, <interface definition>, <signal definition item> \}

FORMALCONTEXTPARAMETER = \{ <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 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 = \{ PREDEFINEDLITERAL, PREDEFINEDOPERATION, PREDEFINEDSORT \}

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

ENTITYDEFINITION = \{ <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, FORMALCONTEXTPARAMETER \}

PREDEFINEDDEFINITION

TYPEREFERENCE = \{ <agent type reference>, <composite state type reference> \}

REFERENCE = \{ <package reference>, <agent reference>, <composite state reference>, <textual operation reference>, TYPEREFERENCE \}

SCOPEUNIT = \{ <package definition>, <agent definition>, <operation definition>, <composite state>, <sort context parameter>, <signal context parameter gen name>, <compound statement>, TYPEDEFINITION \}

2.1.6.3 Function Definitions on AS1

Auxiliary functions on the AS1

The function \texttt{identifier} is used to get the identifier with full qualifier for an entity definition in AS1.

\[
\texttt{identifier}(d; \text{ENTITYDEFINITION}_1): \text{Identifier} = \text{def } \text{mk-} \text{Identifier}(d.\text{fullQualifier}_1, d.\text{entityName}_1)
\]

The function \texttt{fullQualifier} is used to get the full qualifier for an entity definition in AS1.

\[
\text{fullQualifier}(d; \text{ENTITYDEFINITION}_1): \text{Qualifier} = \text{def } \begin{cases} \text{if } \text{su} = \text{undefined} \text{ then empty} \\ \text{else if } d.\text{entityKind} \in \{\text{operation, literal}\} \wedge \text{su} \in \text{Data-type-definition} \text{ then } \text{su}.\text{fullQualifier} \\ \text{else } \text{su}.\text{fullQualifier} \cap \text{mk-Qualifier}(\text{su}.\text{entityKind}, \text{su}.\text{entityName}_1) \end{cases}
\]

\[
\text{idKind}(d; \text{Identifier}): \text{ENTITYKIND}_1 = \text{def } \begin{case} \text{Create-request-node, Signal-destination: agent} \\ \text{Agent-type-definition, Agent-definition: agent type} \\ \text{Procedure-definition, Call-node, Value-returning-call-node: procedure} \\ \text{Gate-definition, Channel-path, Output-node, Save-signalset: signal} \\ \text{Data-type-definition, Parameter, Result, Signal-definition, Timer-definition, Exception-definition, Formal-argument, Variable-definition, Any-expression: sort} \\ \text{Set-node, Reset-node, Timer-active-expression: timer} \\ \text{Originating-gate, Destination-gate: gate} \\ \text{Raise-node: exception} \\ \text{Composite-state-type-definition, State-machine-definition, State-node, State-partition: state type} \\ \text{Literal: literal} \\ \text{Open-range, Operation-application: opration} \\ \text{Input-node:} \\ \text{if } id \text{ in } id.\text{parentASI}.s-\text{Variable-identifier-seq} \text{ then variable} \\ \text{else signal} \\ \text{endif} \\ \text{Handle-node:} \\ \text{if } id \text{ in } id.\text{parentASI}.s-\text{Variable-identifier-seq} \text{ then variable} \\ \text{else exception} \\ \text{endif} \\ \text{Variable-access, Assignment-attempt, Assignment: variable} \\ \text{Direct-via:} \\ \text{if } \text{getEntityDefinition}(id, \text{channel}) \neq \text{undefined} \text{ then channel} \\ \text{else gate} \\ \text{endif} \\ \text{endcase}
\end{case}
\]

The function \texttt{entityName} is used to get the entity name for an entity definition in AS1.

\[
\text{entityName}(d; \text{ENTITYDEFINITION}_1): \text{Name} = \text{def }
\]
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 \( \text{entityKind}_1 \) is used to get the entity kind for an entity definition on AS1.

\[
\text{entityKind}_1(d: \text{ENTITYDEFINITION}_1): \text{ENTITYKIND}_1 = \begin{cases} \\
\text{package} & \text{if Package-definition}\n\text{agent} & \text{if Agent-definition}\n\text{agent type} & \text{if Agent-type-definition}\n\text{procedure} & \text{if Procedure-definition}\n\text{state} & \text{if State-node}\n\text{state type} & \text{if Composite-state-type-definition}\n\text{channel} & \text{if Channel-definition}\n\text{gate} & \text{if Gate-definition}\n\text{signal} & \text{if Signal-definition}\n\text{timer} & \text{if Timer-definition}\n\text{exception} & \text{if Exception-definition}\n\text{variable} & \text{if Variable-definition}\n\text{sort} & \text{if Data-type-definition}\n\text{sort} & \text{if Syntype-definition}\n\text{literal} & \text{if Literal-signature}\n\text{operation} & \text{if Operation-signature}\n\end{cases}
\]

endcase

The function \( \text{getEntityDefinition}_1 \) gets the entity definition for an identifier.

\[
\text{getEntityDefinition}_1(id: \text{Identifier}, k: \text{ENTITYKIND}_1): \text{ENTITYDEFINITION}_1 = \begin{cases} \\
\text{take} & \text{where} \\
(d: \text{ENTITYDEFINITION}_1: d.\text{identifier} = id \land d.\text{entityKind} = k) \rightarrow \\
\text{isActualAndFormalParameterMatched} & (d.\text{actualParameterListOfOpId}, d.\text{formalParameterSortList}))
\end{cases}
\]

The function \( \text{agentKind}_1 \) is used to get the agent kind for an \( \text{Agent-definition} \) or \( \text{Agent-type-definition} \).

\[
\text{agentKind}_1(d: \text{Agent-definition} \cup \text{Agent-type-definition}): \text{AGENTKIND}_1 = \begin{cases} \\
\text{if Agent-definition} \rightarrow d.s-\text{Agent-kind} \land \\
\text{Agent-type-definition} \rightarrow \text{let } td = \text{getEntityDefinition}_1(d.s-\text{Agent-type-identifier}, \text{agent type}) \text{ in} \\\n\text{otherwise undefined} \rightarrow \text{Agent-kind} \land \\
\end{cases}
\]

endlet
endif
The function \( \text{range1} \) returns the set of \( \text{VALUE1} \) corresponding to a Sort.

\[
\text{value}_1(e: \text{Constant-expression}): \text{VALUE1} = \text{def}
\]

\[
\text{case } e \text{ of}
\]

| Literal(*) => compute(e) |
| Conditional-expression(bool, consequence, alternative) => |
| if bool.value, semvalue then consequence.value, else alternative.value, endif |
| Equality-expression(a, b) => computeEquality(a.value1, b.value1) |
| Operation-application(proc, values) => compute(proc, < v in values: v.value1 >) |
| Range-check-expression(range, expr) => expr.value1 \in \text{range1} |

endcase

\( \text{range1} \) returns the set of elements that satisfy the condition.

\[
\text{range1(r: Range-condition): VALUE1-set} = \text{def}
\]

\[
\{ v \in \text{VALUE1}: \exists ci \in r.s-Condition-item:
\]

\[
((ci \in \text{Open-range}) \land \text{isInRange}(ci, ci)) \lor
\]

\[
((ci \in \text{Closed-range}) \land
\]

\[
((ci \in \text{Open-range}) \land \text{isInRange}(ci, ci))\}
\]

\( \text{isInRange}(ci: \text{VALUE1}, r: \text{Open-range}): \text{BOOLEAN} = \text{def} \)

\[
\text{let operator = r.s-Operation-identifier.s-Name in}
\]

\[
\text{let } v' = r.s-\text{Constant-expression.value1 in}
\]

\[
\text{case operator of}
\]

| "v" => v = v' |
| "v" => v \neq v' |
| "v" => v \leq v' |
| "v" => v \geq v' |
| "v" => v > v' |
| "v" => v < v' |

endcase

The function \( \text{staticSort1} \) the static sort of \( e \).

\[
\text{staticSort1(e: Expression): Sort-reference-identifier} = \text{def}
\]

\[
\text{case } e \text{ of}
\]

| Literal => getEntityDefinition(e.s-Literal-identifier, literal).s-Result |
| Variable-identifier => getEntityDefinition(e, variable).s-Sort-reference-identifier |
| Equality-expression \( \lor \) Range-check-expression \( \lor \) Timer-active-expression => |
| \( \text{mk-} \text{Identifier(mk-Qualifier(Name("Predefined")), "Boolean")} \) |
| Now-expression => \( \text{mk-} \text{Identifier(mk-Qualifier(Name("Predefined")), "Time")} \) |
| Pid-expression => \( \text{mk-} \text{Identifier(mk-Qualifier(mk-Name("Predefined")), "Pid")} \) |
| Any-expression => e.s-Sort-reference-identifier |
| Operation-application => getEntityDefinition(e.s-Operation-identifier, operation).s-Result |
| Value-returning-call-node => getEntityDefinition(e.s-Procedure-identifier, procedure).s-Result |
| Conditional-expression => staticSort1(e.s-Consequence-expression) |
| otherwise undefined |

endcase

The predicate \( \text{isDirectSuperType1} \) is used to determine if the first entity definition is the direct super type of the second one.

\[
\text{isDirectSuperType1(d: ENTITYDEFINITION1, d': ENTITYDEFINITION1): BOOLEAN = \text{def}}
\]

\[
\text{case } d' \text{ of}
\]

| TYPEDEFINITION Interface-definition => d = getEntityDefinition(d.s-Identifier, d'.entityKind) |
| Interface-definition => |
| d \in Interface-definition \land |
| (\exists dataId Data-type-identifier: dataId.parentAS1 = d' \land d = getEntityDefinition(dataId, sort)) |
| \text{Syntype-definition} => |
| isDirectSuperType1(d, d'.derivedDataType1) |

ITU-T Z.100/Annex F2 (11/2000) 9
The predicate \( \text{isSuperType}_1 \) is used to determine if the first entity definition is the super type of the second one.

\[
\text{isSuperType}_1(d, \text{ENTITYDEFINITION}_1, d', \text{ENTITYDEFINITION}_1) \triangleq \text{BOOLEAN} \quad \text{def} \\
\text{isDirectSuperType}_1(d, d') \lor \exists \ d'' \in \text{ENTITYDEFINITION}_1: \text{isSuperType}_1 (d, d'') \land \text{isSuperType}_1 (d'', d')
\]

The function \( \text{derivedDataType}_1 \) is used to get the data type definition for a given \( \text{Syntype-definition} \).

\[
\text{derivedDataType}_1(d, \text{Syntype-definition}_1) \cup \text{Data-type-definition}_1 \triangleq \text{Data-type-definition} \quad \text{def} \\
\text{if } d \in \text{Data-type-definition}_1 \text{ then } d \text{ else } \text{getEntityDefinition}_1(d, \text{s-Parent-sort-identifier}, \text{sort}).\text{derivedDataType}_1
\]

The function \( \text{isCompatibleTo}_1 \) determines if a \( \text{Sort-reference-identifier} \) is compatible to the other.

\[
\text{isCompatibleTo}_1(id_1, \text{Sort-reference-identifier} _1, id_2, \text{Sort-reference-identifier}_2) \triangleq \text{BOOLEAN} \quad \text{def} \\
\text{let } d_1 = \text{getEntityDefinition}_1(id_1, \text{sort}) \text{ in} \\
\text{let } d_2 = \text{getEntityDefinition}_1(id_2, \text{sort}) \text{ in} \\
(d_1 = d_2 \lor \text{isSuperType}_1(d_2, d_1)) \text{ endlet}
\]

### 2.1.6.4 Function Definitions on AS0

The function \( \text{name}_0 \) gets the name of a given entity or reference.

\[
\text{name}_0(d, \text{ENTITYDEFINITION}_0 \cup \text{REFERENCE}_0) \triangleq \text{name} \quad \text{def} \\
\text{case } d \text{ of} \\
| \text{REFERENCE}_0 \Rightarrow d, \text{referenceName}_0 \\
| \text{ENTITYDEFINITION}_0 \Rightarrow d, \text{entityName}_0 \\
\text{otherwise undefined} \\
\text{endcase}
\]

The function \( \text{name}_0 \) gets the name of a given entity definition.

\[
\text{name}_0(ed, \text{ENTITYDEFINITION}_0) \triangleq \text{name} \quad \text{def} \\
\text{case } ed \text{ of} \\
| \text{<package definition> } \Rightarrow ed, \text{s-package heading}, \text{s-name} \\
| \text{<system definition> } \Rightarrow ed, \text{s-system heading}, \text{s-name} \\
| \text{<block definition> } \Rightarrow ed, \text{s-block heading}, \text{s-name} \\
| \text{<process definition> } \Rightarrow ed, \text{s-process heading}, \text{s-name} \\
| \text{<procedure definition> } \Rightarrow ed, \text{s-procedure heading}, \text{s-name} \\
| \text{<system type definition> } \Rightarrow ed, \text{s-system type heading}, \text{s-name} \\
| \text{<block type definition> } \Rightarrow ed, \text{s-block type heading}, \text{s-name} \\
| \text{<process type definition> } \Rightarrow ed, \text{s-process type heading}, \text{s-name} \\
| \text{<composite state type definition> } \Rightarrow ed, \text{s-composite state type heading}, \text{s-name} \\
| \text{<textual gate definition> } \Rightarrow ed, \text{s-name} \\
| \text{<textual typebased system definition> } \Rightarrow ed, \text{s-name} \\
| \text{<textual typebased block definition> } \Rightarrow ed, \text{s-name} \\
| \text{<textual typebased process definition> } \Rightarrow ed, \text{s-name} \\
| \text{<textual typebased state partition definition> } \Rightarrow ed, \text{s-name} \\
| \text{<composite state> } \Rightarrow ed, \text{s-composite state heading}, \text{s-name} \\
| \text{<data type definition> } \Rightarrow ed, \text{s-data type heading}, \text{s-name} \\
| \text{<signal definition item> } \Rightarrow ed, \text{s-name} \\
| \text{<exception definition item> } \Rightarrow ed, \text{s-name} \\
| \text{<timer definition item> } \Rightarrow ed, \text{s-name} \\
| \text{<interface definition> } \Rightarrow ed, \text{s-interface heading}, \text{s-name} \\
| \text{<literal signature> } \Rightarrow
\]
if \( ed \in \langle \text{literal name} \rangle \) then \( ed \)
elseif \( ed \in \langle \text{named number} \rangle \) then \( ed.s\in\langle \text{literal name} \rangle \)
else undefined
endif

\[
\text{if } ed.s\text{-implicit} \in \langle \text{operation name} \rangle \text{ then } ed.s\text{-implicit.s}\langle \text{operation name} \rangle \\
\text{else } \text{undefined} \text{ endif}
\]

\[
\text{if } s.e \in \langle \text{syntype definition gen syntype} \rangle \text{ then } s.s\in\langle \text{syntype name} \rangle \\
\text{else } s.s\in\langle \text{data type heading} \rangle s\langle \text{data type name} \rangle \\
\text{endif}
\]

The function \( \text{referenceName}_0 \) gets the name of a given reference.

\[
\text{referenceName}_0(\text{ref}: \text{REFERENCE}_0): \text{name} = \text{def} \\
\text{case } \text{ref} \text{ of} \\
| \langle \text{TyPEREFERENCE}_0 \rangle = d.\text{referenceKind}_0 \\
| \langle \text{textual operation reference} \rangle = \text{ref.s}\langle \text{operation heading} \rangle s\langle \text{operation name} \rangle \\
\text{otherwise } \text{ref.s}\langle \text{name} \rangle \\
\text{endcase}
\]

The function \( \text{kind}_0 \) gets the name of a given entity definition or reference.

\[
\text{kind}_0(d: \text{ENTITYDEFINITION}_0 \cup \text{REFERENCE}_0): \text{ENTITYKIND}_0 = \text{def} \\
\text{case } \text{ed} \text{ of} \\
| \langle \text{REFERENCE}_0 \rangle = d.\text{referenceKind}_0 \\
| \langle \text{ENTITYDEFINITION}_0 \rangle = d.\text{entityKind}_0 \\
\text{endcase}
\]

The function \( \text{entityKind}_0 \) gets the name of a given entity definition.

\[
\text{entityKind}_0(\text{ed}: \text{ENTITYDEFINITION}_0): \text{ENTITYKIND}_0 = \text{def} \\
\text{case } \text{ed} \text{ of} \\
| \langle \text{package definition} \rangle = \langle \text{package} \rangle \\
| \langle \text{system definition} \rangle \cup \langle \text{textual typebased system definition} \rangle = \langle \text{system} \rangle \\
| \langle \text{block definition} \rangle \cup \langle \text{textual typebased block definition} \rangle = \langle \text{block} \rangle \\
| \langle \text{process definition} \rangle \cup \langle \text{textual typebased process definition} \rangle = \langle \text{process} \rangle \\
| \langle \text{system type definition} \rangle = \langle \text{system type} \rangle \\
| \langle \text{block type definition} \rangle = \langle \text{block type} \rangle \\
| \langle \text{process type definition} \rangle = \langle \text{process type} \rangle \\
| \langle \text{composite state} \rangle \cup \langle \text{textual typebased state partition definition} \rangle = \langle \text{state} \rangle \\
| \langle \text{composite state type definition} \rangle \cup \langle \text{composite state type context parameter} \rangle = \langle \text{state type} \rangle \\
| \langle \text{procedure definition} \rangle \cup \langle \text{procedure context parameter} \rangle = \langle \text{procedure} \rangle \\
| \langle \text{signal definition item} \rangle \cup \langle \text{signal context parameter gen name} \rangle = \langle \text{signal} \rangle \\
| \langle \text{data type definition} \rangle \cup \langle \text{syntype definition} \rangle \cup \langle \text{sort context parameter} \rangle = \langle \text{type} \rangle \\
| \langle \text{operation definition} \rangle = \langle \text{ed.s}\langle \text{operation heading} \rangle s\langle \text{operation kind} \rangle \rangle \\
| \langle \text{operation signature} \rangle = \langle \rangle \\
\text{if } \text{ed.parentAS} \in \langle \text{operator list} \rangle \text{ then } \text{operator} \text{ else } \text{method} \text{ endif} \\
| \langle \text{interface definition} \rangle \cup \langle \text{interface context parameter gen name} \rangle = \langle \text{interface} \rangle \\
| \langle \text{textual gate definition} \rangle \cup \langle \text{gate context parameter} \rangle = \langle \text{gate} \rangle \\
| \langle \text{exception definition item} \rangle \cup \langle \text{exception context parameter gen name} \rangle = \langle \text{exception} \rangle \\
| \langle \text{timer definition item} \rangle \cup \langle \text{timer context parameter gen name} \rangle = \langle \text{timer} \rangle \\
| \langle \text{signal list definition} \rangle = \langle \text{signalist} \rangle \\
| \langle \text{literal signature} \rangle = \langle \text{literal} \rangle \\
| \langle \text{agent type context parameter} \rangle \cup \langle \text{agent context parameter} \rangle = \langle \text{ed.}\langle \text{agent kind} \rangle \rangle \\
| \langle \text{remote procedure definition} \rangle \cup \langle \text{remote procedure context parameter} \rangle = \langle \text{remote procedure} \rangle \\
| \langle \text{synonym definition} \rangle \cup \langle \text{synonym context parameter gen name} \rangle = \langle \text{synonym} \rangle
\]
The function \( \text{referenceKind}_0 \) gets the entity kind of a specified reference.

\[
\text{referenceKind}_0(\text{ref}: \text{REFERENCE}_0) : \text{ENTITYKIND}_0 = \text{def}
\]

\[
\begin{align*}
\text{case } \text{ref} \text{ of} \\
| \text{<system type reference> } & \Rightarrow \text{system type} \\
| \text{<block type reference> } & \Rightarrow \text{block type} \\
| \text{<process type reference> } & \Rightarrow \text{process type} \\
| \text{<composite state type reference> } & \Rightarrow \text{state type} \\
| \text{<block reference> } & \Rightarrow \text{block} \\
| \text{<process reference> } & \Rightarrow \text{process} \\
| \text{<composite state reference> } & \Rightarrow \text{state} \\
| \text{<package reference> } & \Rightarrow \text{package} \\
| \text{<signal reference> } & \Rightarrow \text{signal} \\
| \text{<data type reference> } & \Rightarrow \text{type} \\
| \text{<procedure reference> } & \Rightarrow \text{procedure} \\
| \text{<interface reference> } & \Rightarrow \text{interface} \\
| \text{<textual operation reference> } & \Rightarrow \text{ref.s.<operation kind>} \\
\end{align*}
\]

otherwise undefined

endcase

The function \( \text{qualifier}_0 \) gets the qualifier specified in an entity definition or a reference.

\[
\text{qualifier}_0(d: \text{ENTITYDEFINITION}_0 \cup \text{REFERENCE}_0) : \text{<qualifier>} = \text{def}
\]

\[
\text{take}(\{ q \in \text{<qualifier>} : q.\text{parentAS}_0.\text{parentAS}_0 = d \})
\]

The function \( \text{surroundingScopeUnit}_0 \) gets the surrounding scope unit for a node in AS0.

\[
\text{surroundingScopeUnit}_0(d: \text{DefinitionAS}_0) : \text{SCOPEUNIT}_0 = \text{def}
\]

\[
\text{if } d \in \text{<referenced definition> then}
\]

\[
\text{parentAS}_0\text{ofKind}(d.\text{referencedBy}_0, \text{SCOPEUNIT}_0)
\]

\[
\text{else}
\]

\[
\text{parentAS}_0\text{ofKind}(d, \text{SCOPEUNIT}_0)
\]

\end{align*}
\]

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

\[
\begin{align*}
\text{Name} & :: \text{TOKEN} \\
\text{Package-name} &= \text{Name} \\
\text{Agent-type-name} &= \text{Name} \\
\text{Agent-name} &= \text{Name} \\
\text{State-type-name} &= \text{Name} \\
\text{State-name} &= \text{Name} \\
\text{Data-type-name} &= \text{Name} \\
\text{Procedure-name} &= \text{Name} \\
\text{Signal-name} &= \text{Name} \\
\text{Interface-name} &= \text{Name} \\
\text{Literal-name} &= \text{Name} \\
\text{Operation-name} &= \text{Name} \\
\text{Syntype-name} &= \text{Name} \\
\text{Timer-name} &= \text{Name} \\
\text{Gate-name} &= \text{Name} \\
\text{Exception-name} &= \text{Name} \\
\text{Exception-handler-name} &= \text{Name} \\
\text{Connector-name} &= \text{Name} \\
\text{State-entry-point-name} &= \text{Name} \\
\text{State-exit-point-name} &= \text{Name} \\
\text{Channel-name} &= \text{Name} \\
\text{Variable-name} &= \text{Name}
\end{align*}
\]

Concrete Syntax

\[
\begin{align*}
\langle\text{name}\rangle & :: \text{TOKEN} \\
\langle\text{quoted operation name}\rangle & :: \text{TOKEN} \\
\langle\text{character string}\rangle & :: \text{TOKEN} \\
\langle\text{hex string}\rangle & :: \text{TOKEN} \\
\langle\text{bit string}\rangle & :: \text{TOKEN} \\
\langle\text{operation name}\rangle & = \langle\text{operator}\langle\text{name}\rangle | \langle\text{quoted operation name}\rangle \\
\langle\text{literal name}\rangle & = \langle\text{literal}\langle\text{name}\rangle | \langle\text{string name}\rangle \\
\langle\text{string name}\rangle & = \langle\text{character string}\rangle | \langle\text{bit string}\rangle | \langle\text{hex string}\rangle
\end{align*}
\]

Mapping to Abstract Syntax

\[
\begin{align*}
|\langle\text{name}\rangle(x) & \Rightarrow \text{mk-Name}(x) \\
|\langle\text{quoted operation name}\rangle(x) & \Rightarrow \text{mk-Name}(x) \\
|\langle\text{hex string}\rangle(x) & \Rightarrow \text{mk-Name}(x) \\
|\langle\text{bit string}\rangle(x) & \Rightarrow \text{mk-Name}(x) \\
|\langle\text{character string}\rangle(x) & \Rightarrow \\
\text{case } x.\text{parentAS0 of} \\
& | \langle\text{transition option}\rangle \cup \langle\text{task}\rangle \cup \langle\text{answer part}\rangle \cup \langle\text{decision}\rangle \Rightarrow \text{mk-Informal-text}(x) \\
& \text{otherwise } \text{mk-Name}(x) \\
\text{endcase}
\end{align*}
\]

2.2.2 Identifier

Abstract Syntax

\[
\begin{align*}
\text{Identifier} & :: \text{Qualifier Name} \\
\text{Qualifier} & = \text{Path-item}^+ \\
\text{Agent-identifier} & = \text{Identifier} \\
\text{Agent-type-identifier} & = \text{Identifier} \\
\text{Procedure-identifier} & = \text{Identifier}
\end{align*}
\]
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
∀ d, d’ ∈ ENTITYDEFINITION: d.entityKind = d’.entityKind ∧ d ≠ d’ ⇒ d.identifier ≠ d’.identifier

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

Concrete Syntax

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

Conditions on Concrete Syntax
∀ d1, d2 ∈ ENTITYDEFINITION:
(’d1 ≠ d2 ∧ d1.entityKind = d2.entityKind ∧ d1.surroundingScopeUnit = d2.surroundingScopeUnit) ⇒
((d1 ∈ <operation signature> ⇒ d1.entityName ≠ d2.entityName) ∧
(d1 ∈ <operation signature> ⇒ ¬ isSameOperationSignature(d1, d2)) ∨
(d1 ∈ <operation signature> ⇒ d1.entityName ≠ d2.entityName))

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(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(i:<identifier>): <identifier> = i.refersto0.identifier

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

refersto0(i:<identifier>): ENTITYDEFINITION = getEntityDefinition(i, idKind(i))
For any given entity definition in AS0, the function \( \text{identifier}_0 \) returns its identifier with full qualifier.

\[
\text{identifier}_0(\text{def}: \text{ENTITYDEFINITION}_0): \langle \text{identifier} \rangle = \text{def}
\]

\[
\langle \text{identifier} \rangle(\text{def.fullQualifier}_0, \text{def.entityName}_0)
\]

The function \( \text{fullQualifier}_0 \) is used to get the full qualifier for an entity definition.

\[
\text{fullQualifier}_0(d: \text{ENTITYDEFINITION}_0): \langle \text{qualifier} \rangle = \text{def}
\]

\[
\text{let } su = d.\text{surroundingScopeUnit}_0 \text{ in}
\]

\[
\begin{align*}
\text{if } su &= \text{undefined} \text{ then empty} \\
\text{else } su.\text{fullQualifier}_0(\langle \text{path item} \rangle(su.\text{entityKind}_0, su.\text{entityName}_0))
\end{align*}
\]

The function \( \text{getEntityDefinition}_0 \) is used to get the definition that the given identifier refers to.

\[
\text{getEntityDefinition}_0(id: \langle \text{identifier} \rangle, ek: \text{ENTITYKIND}_0): \text{ENTITYDEFINITION}_0 = \text{def}
\]

\[
\begin{align*}
\text{if } ek &\in \{ \text{operator, literal, method} \} \text{ then} \\
\text{resolutionByContext}_0(id)
\end{align*}
\]

\[
\text{else}
\]

\[
\text{let } su = \text{getStartingScopeUnit}_0(id, id.\text{surroundingScopeUnit}_0) \text{ in}
\]

\[
\begin{align*}
\text{if } su &= \text{undefined} \text{ then undefined} \\
\text{else } \text{resolutionByContainer}_0(su, id, ek)
\end{align*}
\]

\[
\text{endif}
\]

The function \( \text{resolutionByContainer}_0 \) binds an \( \langle \text{identifier} \rangle \) to a definition through resolution by container.

\[
\text{resolutionByContainer}_0(su: \text{SCOPEUNIT}_0, id: \langle \text{identifier} \rangle, ek: \text{ENTITYKIND}_0): \text{ENTITYDEFINITION}_0 = \text{def}
\]

\[
\text{let } d1 = \text{bindInLocalDefinition}_0(su, id, ek) \text{ in}
\]

\[
\begin{align*}
\text{if } d1 &= \text{undefined} \text{ then d1} \\
\text{else let } d2 = \text{bindInBaseType}_0(su, id, ek) \text{ in}
\end{align*}
\]

\[
\begin{align*}
\text{if } d2 &= \text{undefined} \text{ then d2} \\
\text{else let } d3 = \text{bindInUsedPackage}_0(su.\langle \text{package use clause} \rangle, id, ek) \text{ in}
\end{align*}
\]

\[
\begin{align*}
\text{if } d3 &= \text{undefined} \text{ then d3} \\
\text{else let } d4 = \text{bindInLocalInterface}_0(su.\text{localInterfaceDefinitionSet}_0, id, ek) \text{ in}
\end{align*}
\]

\[
\begin{align*}
\text{if } d4 &= \text{undefined} \text{ then d4} \\
\text{else let } su' = su.\text{surroundingScopeUnit}_0 \text{ in}
\end{align*}
\]

\[
\begin{align*}
\text{if } su' &= \text{undefined} \text{ then resolutionByContainer}_0(su', id, ek) \\
\text{else undefined}
\end{align*}
\]

\[
\text{endif}
\]

\[
\text{endif}
\]

\[
\text{endif}
\]

\[
\text{endif}
\]

\[
\text{endif}
\]

\[
\text{endif}
\]

\[
\text{endif}
\]

\[
\text{endif}
\]

\[
\text{endif}
\]

\[
\text{endif}
\]

The function \( \text{bindInLocalDefinition}_0 \) is used to search in the given scope unit to determine if there exists a local entity definition for the specified identifier.

\[
\text{bindInLocalDefinition}_0(su: \text{SCOPEUNIT}_0, id: \langle \text{identifier} \rangle, ek: \text{ENTITYKIND}_0): \text{ENTITYDEFINITION}_0 = \text{def}
\]

\[
\text{let } d = \text{take} \{ d : \text{ENTITYDEFINITION}_0 : d.\text{surroundingScopeUnit}_0 = su \wedge \text{isSameEntityName}_0(id.\langle \text{name} \rangle, d) \wedge \text{isConsistentKindTo}_0(d.\text{entityKind}_0, ek) \wedge \text{isVisibleIn}_0(d, id.\text{surroundingScopeUnit}_0) \} \text{ in}
\]

\[
\begin{align*}
\text{if } d &= \text{undefined} \text{ then d} \\
\text{else let } rd = \text{take} \{ rd : \text{REFERENCE}_0 : rd.\text{surroundingScopeUnit}_0 = su \wedge \text{rd.\text{referencedDefinition}.entityName}_0 = id.\langle \text{name} \rangle \wedge \text{isConsistentKindTo}_0(rd.\text{referencedDefinition}_0.\text{entityKind}_0, ek) \wedge \text{isVisibleIn}_0(rd, id.\text{surroundingScopeUnit}_0) \} \text{ in}
\end{align*}
\]
The function \textit{bindInBaseType}, finds the entity definition corresponding to the given \langle identifier \rangle in the base type of the scope unit.

\begin{verbatim}
bindInBaseType\langle su: SCOPEUNIT, id: <identifier>, ek: ENTITYKIND\rangle: ENTITYDEFINITION = def
  let spec = su.specialization\langle in
    if (spec = undefined) \lor isAncestorAS\langle spec, id \rangle then undefined
    else resolutionByContainer\langle spec.s\langle type expression\rangle.baseType, id, ek \rangle
  end
  endif
endlet
endlet
\end{verbatim}

The function \textit{bindInUsedPackage}, finds the entity definition corresponding to the given \langle identifier \rangle in the used packages of the scope unit.

\begin{verbatim}
bindInUsedPackage\langle ucl:<package use clause>*, id: <identifier>, ek: ENTITYKIND\rangle: ENTITYDEFINITION = def
  if ucl = empty then undefined
  elseif ucl.head = id.parentAS\langle then
    bindInUsedPackage\langle ucl.tail, id, ek \rangle
  else
    let d = bindInLocalDefinition\langle ucl.head.usedPackages, id, ek \rangle in
      if d = undefined then d
      else bindInUsedPackage\langle ucl.tail, id, ek \rangle
    endif
  endif
endlet
\end{verbatim}

The function \textit{bindInLocalInterface}, finds the entity definition corresponding to the given \langle identifier \rangle in the interfaces of the scope unit.

\begin{verbatim}
bindInLocalInterface\langle is:<interface definition>-set, id: <identifier>, ek: ENTITYKIND\rangle: ENTITYDEFINITION = def
  if is = \empty then undefined
  else let d = is.take\langle in
    let ed = bindInLocalDefinition\langle d, id, ek \rangle in
      if ed = undefined then ed
      else bindInLocalInterface\langle is \setminus \{d\} \rangle
    endif
  endif
endlet
\end{verbatim}

The function \textit{isSameEntityName}, is used to determine if the given name has the same name as the entity definition.

\begin{verbatim}
isSameEntityName\langle n: <name>, d: ENTITYDEFINITION\rangle: BOOLEAN = def
  (n = d.entityName\langle ) \lor
  (d.entityKind = literal \\land d.e <name class literal> \land
    isInRegularExpr\langle n, d.s\langle regular expression\rangle.s\langle or partial regular expression\rangle-\langle seq\rangle\rangle)
\end{verbatim}

For a given identifier (left most path item may be omitted), the function \textit{getStartingScopeUnit}, gets the starting scope unit denoted by the partial qualifier.

\begin{verbatim}
getStartingScopeUnit\langle id: <identifier>, su: SCOPEUNIT\rangle: SCOPEUNIT = def
  if su = undefined then undefined
  elseif isQualifierMatched\langle id.s\langle qualifier\rangle, su \rangle then su
  else let su1 = getStartingSuInUsedPackage\langle id, su.usedPackageDefinitionList\rangle in
    if su1 = undefined then su1
    else let su2 = getStartingSuInInterface\langle id, su.localInterfaceDefinitionSet\rangle in
      if su2 = undefined then su2
      else getStartingScopeUnit\langle id, su.surroundingScopeUnit\rangle
\end{verbatim}
The function `getStartingSuInUsedPackage` finds the starting scope unit in the packages list.

```plaintext
getStartingSuInUsedPackage(id: <identifier>, pdl:<package definition>*): SCOPEUNIT
if pdl = empty then undefined
elseif isQualifierMatched(id.<qualifier>, pdl.head) then pdl.head
else getStartingSuInUsedPackage(id, pdl.tail)
endif
```

The function `getStartingSuInInterface` finds the starting scope unit in the interface list.

```plaintext
getStartingSuInInterface(id: <identifier>, ifds:<interface definition>-set): SCOPEUNIT
if ifds = ∅ then undefined
else let d = ifds.take in
if isQualifierMatched(id.<qualifier>, d) then d
else getStartingSuInInterface(id, ifds \ {d})
endif
endlet
```

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

```plaintext
isDefinedIn(ed: ENTITYDEFINITION, su: SCOPEUNIT): BOOLEAN
if (su = undefined ∨ ed.undefined) then False
else let su' = ed.surroundingScopeUnit in
isVisibleInInterface(ed, su) ∨
isVisibleInDataType(ed, su) ∨
isVisibleThroughBaseType(ed, su)
endlet
```

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

```plaintext
isVisibleIn(ed: ENTITYDEFINITION, su: SCOPEUNIT): BOOLEAN
isDefinedIn(ed, su) ∨
isVisibleThroughUsedPackage(ed, su) ∨
isVisibleIn(ed, su.surroundingScopeUnit)
```

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

```plaintext
isVisibleInInterface(ed: ENTITYDEFINITION, su: SCOPEUNIT): BOOLEAN
isDefinedIn(ed, su) ∧
isVisibleThroughUsedPackage(ed, su) ∧
isVisibleIn(ed, su.surroundingScopeUnit)
```

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

```plaintext
isVisibleInDataType(ed: ENTITYDEFINITION, su: SCOPEUNIT): BOOLEAN
(isDefinedIn(ed, su) ∧
(isDirectSubType(su, btd) ∧ isVisibleIn(ed, btd)) ∨
(isPublic ed))
```

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

```plaintext
isVisibleThroughBaseType(ed: ENTITYDEFINITION, su: SCOPEUNIT): BOOLEAN
let spec = su.specialization in
if spec = undefined then False
else (∃btd∈ TYPEDEFINITION: isDirectSubType(su, btd) ∧ isVisibleIn(ed, btd)) ∧
The function `isVisibleThroughUsePackage` determines if an entity definition is visible through used packages.

```plaintext
isVisibleThroughUsePackage(ed: ENTITYDEFINITION, su: SCOPEUNIT): BOOLEAN = def
let uc = su.s.<package use clause>.
if uc = ∅ then False
else (∃uc ∈ uc: ed.surroundingScopeUnit = uc.usedPackage ∧ isVisibleThroughUseClause(ed, uc))
endif
endlet
```

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

```plaintext
isBoundToActualContextPara(fcp: FORMALCONTEXTPARAMETER, spec: <specialization>): BOOLEAN = def
(∃acpe <actual context parameter>:
  acpe.parentAS0 = spec.spec
  ∧ isContextParameterCorresponded(fcp, acpe)
)
```

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

```plaintext
isVisibleThroughUseClause(ed: ENTITYDEFINITION, uc:<package use clause>): BOOLEAN = def
let pd = uc.usedPackage.
let pi = pd.s.<package heading>.
if pi = undefined
  (∃ds ∈ <definition selection>: ds.parentAS0 = pi∧ isMentionedInDefSelection(ed, ds, pd)) ∧
  (∃uc ∈ uc: ed.surroundingScopeUnit = uc.usedPackage ∧ isVisibleThroughUseClause(ed, uc))
  ∧ isMentionedInDefSelection(ed, ds, pd)
endif
endlet
```

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

```plaintext
isMentionedInDefSelection(ed: ENTITYDEFINITION, ds:<definition selection>, pd:<package definition>): BOOLEAN = def
(ds.s.<name> = ed.entityName ∧
  (ds.s.<selected entity kind> ≠ undefined ⇒ ds.s.<selected entity kind> = ed.entityKind) ∧
  (ed.entityKind = SIGNAL ∧ ds.s.<selected entity kind> = signal list ∧
    (∃slde <signal list definition>: slde.surroundingScopeUnit = pd ∧
     slde.entityName = ds.s.<name> ∧
     (∃sigld ∈ slde.signalSets: getEntityDefinition(sigld, signal = ed))))
```

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

```plaintext
isConsistentKindTo(t1: ENTITYKIND, t2: ENTITYKIND): 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` is used to determine if the given <qualifier> is the same as the rightmost part of the full <qualifier> denoting the given scope unit.

```plaintext
isQualifierMatched(q: <qualifier>, su: SCOPEUNIT): BOOLEAN = def
if q = undefined then True
else if q ∈ <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

\[
(seq1.length\leq seq2.length\wedge
(\forall i \in 1.. seq1.length\forall j \in NAT: j = seq2.length - seq1.length+ i \Rightarrow
(seq1[i].s.<name> = seq2[j].s.<name> \wedge
(seq1[i].s.<scope unit kind> \neq undefined \Rightarrow
seq1[i].s.<scope unit kind> = seq2[j].s.<scope unit kind>))))
\]
endlet
def

The function \( \text{resolutionByContext}_0 \) is used to bind all \(<\name \rangle \)-s of entity kind \textbf{operator}, \textbf{method} and \textbf{literal} to their corresponding entity definitions.

\[
\text{resolutionByContext}_0(id:\text{<identifier>}:\text{ENTITYDEFINITION}_0) = \text{def}
\]
let \( bl = \text{take(getBindingListSet}_0(id, \text{contextOfIdentifier}_0)) \) in
getDefinitionInBindingList0(id.s.<name>, bl)
endlet
def

The function \( \text{contextOfIdentifier}_0 \) gets the context for resolving the identifier.

\[
\text{contextOfIdentifier}_0(id:\text{<identifier>}:\text{CONTEXT}_0) = \text{def}
\]
if \( \exists \text{exp}\ <\text{expression}>\ : \text{isAncestorAS}_0(\text{exp}, id) \) then
contextOfExp0(parentAS0ofKind(id, <expression>))
else undefined
endlet
def

The function \( \text{getBindingListSet}_0 \) is used to bind all \(<\name \rangle \)-s of entity kind \textbf{operator}, \textbf{method} and \textbf{literal} in the context to their corresponding entity definitions.

\[
\text{getBindingListSet}_0(c: \text{CONTEXT}_0): \text{BINDINGLIST}_0-set = \text{def}
\]
let \( \text{nameList} = c.nameList_0 \) in
let \( \text{possibleBindingListSet} = \text{nameList}\text{possibleBindingListSet}_0 \) in
let \( \text{possibleResultSet} = \{ \text{pbl}\ : \text{possibleBindingListSet}: \text{isSatisfyStaticCondition}0(pbl, c) \} \) in
let \( \text{resultSet} = \{ \text{r}\in \text{possibleResultSet}: \forall \text{r}'\in \text{possibleResultSet}: \text{r} \neq \text{r}' \Rightarrow
\text{mismatchNumber}_0(\text{r}, c)\leq \text{mismatchNumber}_0(\text{r}', c) \} \) in
if \( \text{resultSet} = 1 \) then \text{resultSet}
else \( \text{resultSet} = 0 \) then \( \emptyset \)
else \( (\forall \text{bl}1, \text{bl}2\in \text{resultSet}: \forall i \in 1.. \text{bl}1.length: \text{bl}1[i].s.<\text{ENTITYDEFINITION}_0\.\text{entityKind}_0\neq \text{literal} \Rightarrow
\text{representSameDynamicOpSig}_0(\text{bl}1[i].s.<\text{ENTITYDEFINITION}_0, \text{bl}2[i].s.<\text{ENTITYDEFINITION}_0)) \)
then \text{resultSet}
else \( \emptyset \)
endlet
def

\[
\text{representSameDynamicOpSig}_0(\text{os}1:.\text{<operation signature>}, \text{os}2:.\text{<operation signature>}:\text{BOOLEAN} = \text{def}
(\text{os}1.\text{superCounterpart}_0 \cup \{ \text{os}1 \} \cap (\text{os}2.\text{superCounterpart}_0 \cup \{ \text{os}2 \}) \neq \emptyset
\]
def

The function \( \text{nameList}_0 \) gets all the \(<\name \rangle \)-s of entity kind \textbf{operator}, \textbf{method} and \textbf{literal} appeared in the context.

\[
\text{nameList}_0(c: \text{CONTEXT}_0): <\name >* = \text{def}
\]
case c of
\(<\text{assignment}>\Rightarrow c.s.<\text{variable}>\text{nameListInVariable}_0\ c.s.<\text{expression}>\text{nameListInExpression}_0
\(<\text{decision}>\Rightarrow c.s.<\text{nameListInDecision}_0
\(<\text{expression}>\Rightarrow c.s.<\text{nameListInExpression}_0
otherwise \text{empty}
endcase
def

The function \( \text{nameListInExpression}_0 \) gets all the \(<\name \rangle \)-s of entity kind \textbf{operator}, \textbf{method} and \textbf{literal} appeared in the \(<\text{expression}>\).
nameListInExpression(expr: <expression>):<name>*=def
  case expr of
  |<create expression><value returning procedure call>=>
    expr.actualParameterList0.nameListInActualParameterList0
  |<range check expression>=><expr.s<expression>.nameListInExpression0
  |<binary expression>(e1, op, e2) =>
    e1.nameListInExpression0.op.e2.nameListInExpression0
  |<equality expression>=><expr.s<expression>.nameListInExpression0
  |<expression gen primary>=><expr.s-implicit<expr.s<primary>.nameListInPrimary0
    endcase

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

nameListInPrimary0(p: <primary>):<name>*=def
  case p of
  |<operator application>=>p.nameListInOperationApplication0
  |<literal>=><p.s-literal identifier>.s-literal name>
  |<expression>=><p.nameListInExpression0
  |<conditional expression>=><p.s-expression>.nameListInExpression0
    p.s<consequence expression>.nameListInExpression0
    p.s<alternative expression>.nameListInExpression0
  |<spelling term>=><p.s<name>>
  |<extended primary>=><p.nameListInExtendedPrimary0
    otherwise: empty
    endcase

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

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

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

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

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

nameListInActualParameterList0(el: <expression>*):<name>*=def
  if el = empty then empty
  else
    el.head.nameListInExpression0+el.tail.nameListInActualParameterList0
  endif
The function \textit{nameListInVariable} gets all the \texttt{name}s of entity kind \texttt{operator}, \texttt{method} and \texttt{literal} appeared in \texttt{<variable>}.

\begin{verbatim}
nameListInVariable(v:<variable>):<name>*=def
  if v <: indexed variable> then
    v.s.<variable>.nameListInVariable0 v.s.<actual parameter>-seq.nameListInActualParameterList0
  elsif v <: field variable> then
    v.s.<variable>.nameListInVariable0
  else empty endif
\end{verbatim}

The function \textit{nameListInDecision} gets all the \texttt{name}s of entity kind \texttt{operator}, \texttt{method} and \texttt{literal} appeared in the \texttt{<decision>}.

\begin{verbatim}
nameListInDecision(d:<decision>):<name>*=def
d.s.<question>.nameListInExpression0 \cap d.rangeConditionList0.nameListInRangeConditions0
\end{verbatim}

The function \textit{nameListInRangeConditions} gets all the \texttt{name}s of entity kind \texttt{operator}, \texttt{method} and \texttt{literal} appeared in the \texttt{<range condition>} list.

\begin{verbatim}
nameListInRangeConditions(rcl:<range condition>*):<name>*=def
  if rcl = empty then empty
  else rcl.head.nameListInRangeList0 rcl.tail.nameListInRangeConditions0
\end{verbatim}

The function \textit{nameListInRangeList} gets all the \texttt{name}s of entity kind \texttt{operator}, \texttt{method} and \texttt{literal} appeared in the \texttt{<range>} list.

\begin{verbatim}
nameListInRangeList(rl:<range>*):<name>*=def
  if rl = empty then empty
  else rl.head.nameListInRange0 rl.tail.nameListInRangeList0
\end{verbatim}

The function \textit{nameListInRange} gets all the \texttt{name}s of entity kind \texttt{operator}, \texttt{method} and \texttt{literal} appeared in the \texttt{<range>}.

\begin{verbatim}
nameListInRange(r:<range>):<name>*=def
  case r of
    |<closed range>(c1, c2)=>
      c1.nameListInExpression0 \cap c2.nameListInExpression0
    |<open range>(c1, n, c2)=>
      c1.nameListInExpression0 \cap n \cap c2.nameListInExpression0
  endcase
\end{verbatim}

Each element in the \textit{possibleBindingListSet} represents a possible resolution for the given name list.

\begin{verbatim}
possibleBindingListSet(n:<name>*):BINDINGLIST0-set =def
  \{b \in BINDINGLIST0: b.length = n.length ∧
  \forall i \in 1.. b.length: b[i].s.<name> = n[i] ∧ b[i].s.ENTITYDEFINITION \in n[i].possibleDefinitionSet0\}
\end{verbatim}

The function \textit{isSatisfyStaticCondition} determines if the binding violates any static sort constraints in the context.

\begin{verbatim}
isSatisfyStaticCondition(bl:BINDINGLIST0, c:CONTEXT0): BOOLEAN0=def
  case c of
    |<assignment>=> isSatisfyAssignmentCondition0(bl, c)
    |<decision>=> isSatisfyDecisionCondition0(bl, c)
    |<expression>=> isSatisfyExpressionCondition0(bl, c)
    otherwise False endcase
\end{verbatim}

The function \textit{isSatisfyAssignmentCondition} determines if the binding violates any static sort constraints in the \texttt{<assignment>}.

\begin{verbatim}
isSatisfyAssignmentCondition0(bl:BINDINGLIST0, ass: <assignment>): BOOLEAN0=def
\end{verbatim}
let varSort = getVariableSort(ass.s.<variable>) in
let expSort = getStaticSort(ass.s.<expression>, bl) in
(isSortCompatible(varSort, expSort) \(\lor\) isSortCompatible(expSort, varSort))\(\land\)
(ass.s.<variable>\(\in\) <indexed variable>\(\Rightarrow\)isSatisfyIndexVariableCondition(bl, ass.s.<variable>))
endlet

The function isSatisfyIndexVariableCondition\(_b\) determines if the binding violates any static sort constraints in the <indexed variable>.

isSatisfyIndexVariableCondition(bl, var.<indexed variable>): BOOLEAN = def
let acp = var.s.<actual parameter>-seq in
isSortCompatible(getStaticSort(acp[1], bl), getIndexSort(getVariableSort(var)))\(\land\)
(var.s.<variable>\(\in\) <indexed variable>\(\Rightarrow\)isSatisfyIndexVariableCondition(bl, var.s.<variable>))
endlet

Get the static sort of a <variable>.

getVariableSort(var.<variable>):<sort> = def
\begin{cases}
\text{case var of }
| <identifier>=\Rightarrow getEntityDefinition(var, variable), parentAS1.s.<sort>
| <indexed variable>=\Rightarrow getItemSort(getVariableSort(var))
| <field variable>=\Rightarrow getFieldSort(getVariableSort(var.<variable>), var.s.<field name>)
endcase
\end{cases}

The function getItem\(_s\) gets the item sort of a <sort> that is a subtype of the predefined sort String or Array.

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

The function getIndex\(_s\) gets the index sort of a <sort> that is a subtype of the predefined sort String or Array.

getIndexSort(s.<sort>):<sort> = def
let d = getEntityDefinition(s, sort).derivedDateType in
if d.specialization\(_0\) = undefined then undefined
else if d.specialization\(_0\).s.<base type>.s.<name> = "String" then
<identifier>(<qualifier>(<name>="Predefined"), "Integer")
else if d.specialization\(_0\).s.<base type>.s.<name> = "Array" then d.actualContextParameterList[1]
else undefined
endif
endlet

The function getField\(_s\) gets the field sort of a field name in the <data type definition> referred by the given <sort>.

g.getFieldSort(s.<sort>, n.<name>):<sort> = def
let d = getEntityDefinition(s, sort).derivedDateType in
if d.s.<data type definition body>.s.<data type constructor> in
\begin{cases}
\text{if cons = }<\text{structure definition} \Rightarrow\text{then}
take [fos.s.<sort>: fos\in<fields of sort>\&(\exists\text{name}<\text{name}: name.parentAS0=fos\text{name}=n})]
else undefined
endif
\end{cases}
endlet

The function isSatisfyStaticCondition\(_b\) determines if the binding violates any static sort constraints in the <decision>.

isSatisfyDecisionCondition\(_b\)(bl.BINDINGLIST\(_b\), d.<decision>): BOOLEAN = def
let q = d.s.<question>, rcs = d.rangeConditionList\(_b\).toSet in
The function \( \text{isSatisfyExpressionCondition0} \) determines if the binding violates any static sort constraints in the <expression>.

\[
\text{isSatisfyExpressionCondition0}(\text{bl}: \text{BINDINGLIST0}, \text{exp}: \text{<expression>}): \text{BOOLEAN} = \text{def}
\]

case exp of
|<create expression>=|=\text{isSatisfyCreateCondition0}(\text{bl}, \text{exp})
|<value returning procedure call>=|=\text{if exp \text{<procedure call body}> then isSatisfyProcedureCallBodyCondition0(\text{bl}, \text{exp}) else isSatisfyRemoteProcCallBodyCondition0(\text{bl}, \text{exp}) endif}
|<range check expression>=|=\text{isSatisfyRangeCheckCondition0(\text{bl}, \text{exp})}
|<equality expression>=|=\text{isSatisfyEqualityExpCondition0(\text{bl}, \text{exp})}
|<binary expression>=|=\text{let opDef = getDefinitionInBindingList(exp.s-implicit, \text{bl}) in}
\quad \text{let fpl = opDef.operationParameterSortList0 in}
\quad \text{fpl.length = 2∧}
\quad \text{isSortCompatible0(getStaticSort0(\text{exp.s-<expression>}, \text{bl}), fpl[1])∧}
\quad \text{isSortCompatible0(getStaticSort0(\text{exp.s2-<expression>}, \text{bl}), fpl[2])∧}
\quad \text{isSatisfyExpressionCondition0(\text{bl}, \text{exp.s-<expression>}))∧}
\quad \text{isSatisfyExpressionCondition0(\text{bl}, \text{exp.s2-<expression>))}
\quad \text{endlet}
|<expression gen primary>=|=\text{if exp.s-implicit = undefined then}
\quad \text{isSatisfyPrimaryCondition0(\text{bl}, \text{pr}) else}
\quad \text{let opDef = getDefinitionInBindingList(exp.s-implicit, \text{bl}) in}
\quad \text{let fpl = opDef.operationParameterSortList0 in}
\quad \text{fpl.length = 1∧isSortCompatible0(getStaticSort0(\text{pr}, \text{bl}), fpl[1])∧}
\quad \text{isSatisfyPrimaryCondition0(\text{bl}, \text{pr})}
\quad \text{endlet}
\quad \text{endif endlet}

The function \( \text{isSatisfyCreateCondition0} \) determines if the binding violates any static sort constraints in the <expression>.

\[
\text{isSatisfyCreateCondition0}(\text{bl: BINDINGLIST0}, \text{ce}: \text{<create expression>}): \text{BOOLEAN} = \text{def}
\]

let def = ce.getCreatedAgentDefinition0 in
\quad \text{if def = undefined then False else}
\quad \text{let fpl = def.agentFormalParameterList0 in}
\quad \text{let apl = ce.actualParameterList0 in}
\quad \text{fpl.length = apl.length ∧}
\quad \text{(\(\forall i \in 1..\text{fpl.length}: \text{isSortCompatible0(getStaticSort0(\text{apl}[i], \text{bl}), fpl[i].parentASI.s-<sort>))∧
\quad \text{isSatisfyExpressionCondition0(\text{bl}, \text{apl}[i])})
\quad \text{endlet}
\quad \text{endif endlet
The function `getCreateExpSort0` gets the static sort of `<create expression>`.

```
getCreateExpSort0(ce: <create expression>): <sort> = def
  let def = ce.getCreatedAtDefinition0 in
  if def = undefined then Pid
  else def.identifier0
end endlet
```

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

```
isSatisfyProcedureCallBodyCondition0(bl: BINDINGLIST0, body: <procedure call body>): BOOLEAN = def
  let apl = body.actualParameterList0 in
  let fpsl = calledProcedure0.formalParameterSortList0 in
  fpsl.length = apl.length ∧
  (\forall i ∈ 1..fpsl.length: isSortCompatible0(getStaticSort0(apl[i], bl), fpsl[i]) ∧
  isSatisfyExpressionCondition0(bl, apl[i]))
end
```

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

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

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

```
procedureResult0(pd: <procedure definition>): <sort> = def
  if pd.s.<procedure heading>.s.<procedure result> ≠ undefined then
    pd.s.<procedure heading>.s.<procedure result>
  elseif pd.specialization0 ≠ undefined then
    pd.baseType0.procedureResult0
  else undefined
endif
```

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

```
getRemoteProcCallBodySort0(body: <remote procedure call body>, bl: BINDINGLIST0): <sort> = def
  getEntityDefinition0(body.s.<remote procedure <identifier>, remote procedure).procedureResult0
```

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

```
isSatisfyRemoteProcCallBodyCondition0(bl: BINDINGLIST0, body: <remote procedure call body>): BOOLEAN = def
  let rpd = getEntityDefinition0(body.s.<remote procedure <identifier>, remote procedure) in
  let fpsl = rpd.formalParameterSortList0 in
  let apl = body.actualParameterList0 in
  fpsl.length = apl.length ∧
  (\forall i ∈ 1..fpsl.length: isSortCompatible0(getStaticSort0(apl[i], bl), fpsl[i]) ∧
  isSatisfyExpressionCondition0(bl, apl[i]))
end
```

The function `operationResult0` gets the result sort of an `<operation signature>`.

```
operationResult0(os: <operation signatures>): <sort> = def
  if os ∈ PREDEFINEDOPERATION0 then os.getPredefinedOpResult0
  else os.s.<result>
endif
```
The function \texttt{isSatisfyRangeCheckCondition}_0 determines if the binding violates any static sort constraints in the \texttt{<range check expression>}.

\begin{verbatim}
isSatisfyRangeCheckCondition\(_0(bl: BINDINGLIST, rce: \texttt{<range check expression>})\): BOOLEAN=\begin{def}
if \texttt{rce.s-implicit}e <\texttt{sort}> then
  isSatisfyExpressionCondition\(_0(bl, rce.s-\texttt{expression})\)∧
  isSameSort\(_0\)(getStaticSort\(_0\)(rce.s-\texttt{expression}, bl), rce.s-\texttt{implicit.s-}\texttt{sort})
else
  isSatisfyExpressionCondition\(_0(bl, rce.s-\texttt{expression})\)∧
  isSameSort\(_0\)(getStaticSort\(_0\)(\texttt{<primary>}, bl), rce.s-\texttt{implicit.s-}\texttt{sort})
endif
\end{def}
\end{verbatim}

The function \texttt{isSatisfyEqualityExpCondition}_0 determines if the binding violates any static sort constraints in the \texttt{<equality expression>}.

\begin{verbatim}
isSatisfyEqualityExpCondition\(_0(bl: BINDINGLIST, eq: \texttt{<equality expression>})\): BOOLEAN=\begin{def}
isSatisfyExpressionCondition\(_0(bl, eq.s-\texttt{expression})\)∧
isSatisfyExpressionCondition\(_0(bl, eq.s2-\texttt{expression})\)∧
(isSortCompatible\(_0\)(getStaticSort\(_0\)(eq.s-\texttt{expression}, bl), getStaticSort\(_0\)(eq.s2-\texttt{expression}, bl))\)
\end{def}
\end{verbatim}

The function \texttt{isSatisfyPrimaryCondition}_0 determines if the binding violates any static sort constraints in the \texttt{<primary>}.

\begin{verbatim}
isSatisfyPrimaryCondition\(_0(bl: BINDINGLIST, pr: \texttt{<primary>})\): BOOLEAN=\begin{def}
case pr of
|\texttt{<operator application>}=\Rightarrow\texttt{isSatisfyOpAppCondition}\(_0(bl, pr)\)
|\texttt{<method application>}=\Rightarrow\texttt{isSatisfyMethodAppCondition}\(_0(bl, pr)\)
|\texttt{<expression>}=\Rightarrow\texttt{isSatisfyExpressionCondition}\(_0(bl, pr)\)
|\texttt{<conditional expression>}=\Rightarrow
  isSatisfyExpressionCondition\(_0(bl, pr.s-\texttt{expression})\)∧
  isSatisfyExpressionCondition\(_0(bl, pr.s-\texttt{consequence expression})\)∧
  isSatisfyExpressionCondition\(_0(bl, pr.s-\texttt{alternative expression})\)
|\texttt{<extended primary>}=\Rightarrow\texttt{isSatisfyExtendedPrimaryCond}\(_0(bl, pr)\)
otherwise True
endcase
\end{def}
\end{verbatim}

The function \texttt{isSatisfyExtendedPrimaryCond}_0 determines if the binding violates any static sort constraints in the \texttt{<extended primary>}.

\begin{verbatim}
isSatisfyExtendedPrimaryCond\(_0(bl: BINDINGLIST, epr: \texttt{<extended primary>})\): BOOLEAN=\begin{def}
case epr of
|\texttt{<indexed primary>}=\Rightarrow
  \forall i \in 1..\texttt{seq.length}:
  isSortCompatible\(_0\)(getStaticSort\(_0\)(\texttt{seq}[i]), bl)
  getIdxSort\(_0\)(getPrimarySort\(_0\)(epr.s-\texttt{primary}), bl))
|\texttt{<field primary>}=\Rightarrow
  (epr.s-\texttt{primary} ≠ undefined)⇒
  isSatisfyPrimaryCondition\(_0\)(epr.s-\texttt{primary}, bl)\∧
  getFieldSort\(_0\)(getPrimarySort\(_0\)(epr.s-\texttt{primary}, bl), epr.s-\texttt{field name}) ≠ undefined
|\texttt{<composite primary>}=\Rightarrow
  let sl = getStaticSort\(_0\)(\texttt{para}, bl) para in epr.s-\texttt{actual parameter}-\texttt{seq} empty⇒
  (getCompositeSort\(_0\)(sl) ≠ undefined ∧
  \forall i \in 1..\texttt{seq.length}:
  epr.s-\texttt{actual parameter}-\texttt{seq}[i] ≠ undefined⇒
  isSatisfyExpressionCondition\(_0\)(bl, epr.s-\texttt{actual parameter}-\texttt{seq}[i]))
endlet
\end{def}
\end{verbatim}

The function \texttt{getCompositeSort}_0 gets the sort that refers to a structure data type whose field sort list is the same as the specified parameters.
The function `getPrimarySort` gets the static sort of an `<primary>`.

```plaintext
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.
sorted.0
    |<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)
    |<variable access>=> getVarAccessSort0(pr)
    |<imperative expression>=> getImperativeExpSort0(pr)
    |<synonym>=> getSynonymSort0(pr)
    otherwise True
  endcase
```

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

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

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

```plaintext
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-
    |<pid expression>=>
      if ie ≠ self then <identifier>(<qualifier>(<name>("Predefined")), "Pid")
      else
        let def = parentAS0ofKind(ie, <agent definition>.
      if def ≠ undefined then <identifier>(<qualifier>(<name>("Predefined")), "Pid")
        else def.identifier0
      endlet
    endif
    |<any expression>=> ie.s-
    |<state expression>=> <identifier>(<qualifier>(<name>("Charstring")))
  endcase
```

The function `getSynonymSort0` gets the static sort of an `<synonym>`. **
The function \texttt{isSatisfyOpAppCondition}_0 determines if the binding violates any static sort constraints in the \texttt{<operator application>}. 

\[
\texttt{isSatisfyOpAppCondition}_0(b|l: \texttt{BINDINGLIST}_0, o|a: \texttt{<operation application>}): \texttt{BOOLEAN} = \texttt{def}
\]

\[
\begin{align*}
\text{let } & \texttt{opDef} = \text{getDefinitionInBindingList}(a|o, s|<\text{operation identifier}>=s|<\text{name}>, b|l) \text{ in} \\
& \texttt{let } \texttt{fpl} = \texttt{opDef} \texttt{.operationParameterSortList}_0 \texttt{in} \\
& \texttt{let } \texttt{apl} = a|o \texttt{.actualParameterList}_0 \texttt{in} \\
& \texttt{fpl} \texttt{.length} = \texttt{apl} \texttt{.length} \land \\
& (\forall i \in 1..\texttt{fpl} \texttt{.length}: \texttt{isSortCompatible}_0(\texttt{getStaticSort}(\texttt{apl}[i], b|l), \texttt{fpl}[i]) \land \\
& \texttt{isSatisfyExpressionCondition}_0(\texttt{bl}, \texttt{apl}[i])) \\
& \text{endl} \\
\end{align*}
\]

The function \texttt{getOpAppSort}_0 gets the static sort of an \texttt{<synonym>}. 

\[
\texttt{getOpAppSort}_0(o|a: \texttt{<operation application>}, b|l: \texttt{BINDINGLIST}_0): \texttt{<sort>} = \texttt{def}
\]

\[
\begin{align*}
\text{getDefinitionInBindingList}(a|o, s|<\text{operation identifier}>=s|<\text{name}>, b|l) \text{.operationResult}_0 \\
\end{align*}
\]

The function \texttt{isSatisfyMethodAppCondition}_0 determines if the binding violates any static sort constraints in the \texttt{<method application>}. 

\[
\texttt{isSatisfyMethodAppCondition}_0(b|l: \texttt{BINDINGLIST}_0, m|a: \texttt{<method application>}): \texttt{BOOLEAN} = \texttt{def}
\]

\[
\begin{align*}
\text{let } & \texttt{opDef} = \text{getDefinitionInBindingList}(a|m, s|<\text{operation identifier}>=s|<\text{name}>, b|l) \text{ in} \\
& \texttt{let } \texttt{fpl} = \texttt{opDef} \texttt{.operationParameterSortList}_0 \texttt{in} \\
& \texttt{let } \texttt{apl} = a|m \texttt{.actualParameterList}_0 \texttt{in} \\
& \texttt{fpl} \texttt{.length} = \texttt{apl} \texttt{.length} \land \texttt{isSatisfyPrimaryCondition}_0(b|l, m|a|s|<\text{primary}>)) \land \\
& (\forall i \in 1..\texttt{fpl} \texttt{.length}: \texttt{isSortCompatible}_0(\texttt{getStaticSort}(\texttt{apl}[i], b|l), \texttt{fpl}[i]) \land \\
& \texttt{isSatisfyExpressionCondition}_0(\texttt{bl}, \texttt{apl}[i]) \\
& \text{endl} \\
\end{align*}
\]

The function \texttt{operationParameterSortList}_0 gets the operation formal parameter sort list. 

\[
\texttt{operationParameterSortList}_0(o|s: \texttt{<operation signature>}): \texttt{<sort>} = \texttt{def}
\]

\[
\begin{align*}
\text{if } & \texttt{o|s} \texttt{.getPredefinedOpParas}_0 \neq \texttt{undefined} \text{ then} \\
& \texttt{o|s} \texttt{.getPredefinedOpParas}_0 \\
\text{else} & \text{<paras s|<formal parameter>: s|<sort>} | \texttt{paras in} \texttt{o|s} \texttt{.operationSignatureParameterList}_0 \\
& \text{endif} \\
\end{align*}
\]

The function \texttt{getDefinitionInBindingList}_0 gets the corresponding entity definition for a name in a binding list. 

\[
\texttt{getDefinitionInBindingList}_0(n|<\text{name}>, b|l: \texttt{BINDINGLIST}_0): \texttt{ENTITYDEFINITION}_0 = \texttt{def}
\]

\[
\begin{align*}
\text{take} \{ \texttt{d|e|ENTITYDEFINITION}_0 : \exists i \in 1..\texttt{bl} \texttt{.length}: \texttt{bl}[i].s|<\text{name}>=n \land \texttt{bl}[i].s|\texttt{ENTITYDEFINITION}_0=d \} \\
\end{align*}
\]

The function \texttt{possibleDefinitionSet}_0 gets the set of possible entity definition for a name. 

\[
\texttt{possibleDefinitionSet}_0(n|<\text{name}>): \texttt{ENTITYDEFINITION}\_0 \texttt{set} = \texttt{def}
\]

\[
\begin{align*}
\{ \texttt{d|e|ENTITYDEFINITION}_0 : (\exists \texttt{d|e|ENTITYDEFINITION}_0.(\texttt{d|e|ENTITYDEFINITION}_0 != n)) \land \\
& ((\texttt{d|e|ENTITYDEFINITION}_0.(\texttt{d|e|ENTITYDEFINITION}_0 == n)) \lor \\
& (\texttt{d|e|ENTITYDEFINITION}_0.(\texttt{d|e|ENTITYDEFINITION}_0 == n) \land \texttt{d|e|ENTITYDEFINITION}_0.(\texttt{d|e|ENTITYDEFINITION}_0 == \texttt{d|e|ENTITYDEFINITION}_0))) \\
& (\texttt{d|e|ENTITYDEFINITION}_0.(\texttt{d|e|ENTITYDEFINITION}_0 == \texttt{PREDEFINEDOPERATION}) \land \texttt{d|e|ENTITYDEFINITION}_0.(\texttt{d|e|ENTITYDEFINITION}_0 == \texttt{LITERAL}) \lor \\
& (\texttt{d|e|ENTITYDEFINITION}_0.(\texttt{d|e|ENTITYDEFINITION}_0 == \texttt{PREDEFINEDOPERATION}) \land \texttt{d|e|ENTITYDEFINITION}_0.(\texttt{d|e|ENTITYDEFINITION}_0 == \texttt{LITERAL})) \} \\
\end{align*}
\]

The function \texttt{isRenamedBy}_0 determines if a name is renamed by another one. 

\[
\texttt{isRenamedBy}_0(n_1 , n_2 |<\text{name}>): \texttt{BOOLEAN} = \texttt{def}
\]

\[
(\exists \texttt{rp|e|<rename pair>}.(\texttt{rp|e|<operation name> = n_2} \land \texttt{rp|e|<operation name> = n_1}) \lor \\
(\texttt{rp|e|<literal name> = n_2} \land \texttt{rp|e|<literal name> = n_1}) \lor \\
\]
\( (\exists n3 \in \text{name} : \text{isRenamedBy}_{0}(n1, n3) \land \text{isRenamedBy}_{0}(n3, n2)) \)

The function \( \text{actualParameterList}_{0} \) gets the actual parameter list for a \(<\text{create expression}>\), a \(<\text{value returning procedure call}>\) or an \(<\text{operation application}>\).

\[
\text{actualParameterList}_{0}(d) : \text{<create expression>} \cup \text{<value returning procedure call>} \cup \text{<operation application>}) = \begin{cases} 
    d. \ s \rightarrow \text{actual parameter} & \text{seq}
\end{cases}
\]

The function \( \text{getCreatedAgentDefinition}_{0} \) gets the \(<\text{agent definition}>\) or \(<\text{agent type definition}>\) that the \(<\text{create expression}>\) involves.

\[
\text{getCreatedAgentDefinition}_{0}(ce) : \text{<create expression>} = \begin{cases} 
    \text{id} = \begin{cases} 
        \text{id.\ s-implicit} & \text{in } \text{if } \text{id} \in \text{<identifier>} \text{ then } \text{getEntityDefinition}_{0}(\text{id, id.idKind}_{0}) \\
        \text{undefined} & \text{elseif } \text{id.\ surroundingScopeUnit}_{0} \in \text{<agent definition>} \cup \text{<agent type definition>} \text{ then } \text{id.\ surroundingScopeUnit}_{0} \\
        \text{else undefined} & \text{endif}
    \end{cases}
\end{cases}
\]

The function \( \text{getStaticSort}_{0} \) gets the static sort of an expression.

\[
\text{getStaticSort}_{0}(exp) : \text{<expression>} = \begin{cases} 
    \text{case } exp \text{ of } \begin{cases} 
        \text{<create expression>} & \text{getCreateExpSort}_{0}(exp) \\
        \text{<value returning procedure call>} & \begin{cases} 
            \text{if } exp \in \text{<procedure call body>} \text{ then } \text{getProcCallBodySort}_{0}(exp) \\
            \text{else } \text{getRemoteProcCallBodySort}_{0}(exp, bl) & \text{endif}
        \end{cases}
    \end{cases}
\end{cases}
\]

The function \( \text{mismatchNumber}_{0} \) counts the number of mismatches that the static sort of an \(<\text{expression}>\) is not the same as the static sort of the \(<\text{variable}>\) or the static sort of an actual parameter is not the same as the sort of the corresponding formal parameter.

\[
\text{mismatchNumber}_{0}(bl) : \text{<BINDINGLIST}_{0}, c : \text{<CONTEXT>}_{0}) = \begin{cases} 
    \text{case } c \text{ of } \begin{cases} 
        \text{<assignment>} & \text{mismatchNumberOfAssignment}_{0}(bl, c) \\
        \text{<decision>} & \text{mismatchNumberOfDecision}_{0}(bl, c) \\
        \text{<expression>} & \text{mismatchNumberOfExpression}_{0}(bl, c)
    \end{cases}
\end{cases}
\]

The function \( \text{mismatchNumberOfAssignment}_{0} \) gets the mismatch number of the variable.

\[
\text{mismatchNumberOfAssignment}_{0}(bl) : \text{<BINDINGLIST}_{0}, ass : \text{<assignment>}) = \begin{cases} 
    \text{let } \text{varSort} = \text{getVariableSort}_{0}(ass.\ s-<\text{variable}) \text{ in } \text{opDef.operationResult}_{0}
\end{cases}
\]

The function \( \text{mismatchNumberOfExpression}_{0} \) gets the mismatch number of the expression.

\[
\text{mismatchNumberOfExpression}_{0}(bl) : \text{<BINDINGLIST}_{0}, ass : \text{<expression>, bl} = \begin{cases} 
    \text{case } ass \text{.s-<variable> of } \begin{cases} 
        \text{<identifier>} & \text{mismatchNumber}_{0}(bl) \\
        \text{<field variable>} & \text{mismatchNumber}_{0}(bl)
    \end{cases}
\end{cases}
\]

\[
\text{mismatchNumber}_{0}(bl) = \begin{cases} 
    \text{case } c \text{ of } \begin{cases} 
        \text{<assignment>} & \text{mismatchNumberOfAssignment}_{0}(bl, c) \\
        \text{<decision>} & \text{mismatchNumberOfDecision}_{0}(bl, c) \\
        \text{<expression>} & \text{mismatchNumberOfExpression}_{0}(bl, c)
    \end{cases}
\end{cases}
\]

The function \( \text{mismatchNumberOfDecision}_{0} \) gets the mismatch number of the decision.

\[
\text{mismatchNumberOfDecision}_{0}(bl) : \text{<BINDINGLIST}_{0}, ass : \text{<decision>)} = \begin{cases} 
    \text{let } \text{varSort} = \text{getVariableSort}_{0}(ass.\ s-<\text{variable}) \text{ in } \text{opDef.operationResult}_{0}
\end{cases}
\]

The function \( \text{mismatchNumberOfExpression}_{0} \) gets the mismatch number of the expression.

\[
\text{mismatchNumberOfExpression}_{0}(bl) : \text{<BINDINGLIST}_{0}, ass : \text{<expression>, bl} = \begin{cases} 
    \text{case } ass \text{.s-<variable> of } \begin{cases} 
        \text{<identifier>} & \text{mismatchNumber}_{0}(bl) \\
        \text{<field variable>} & \text{mismatchNumber}_{0}(bl)
    \end{cases}
\end{cases}
\]
if \( \neg \text{isSameSort}(\text{varSort}, \text{expSort}) \) then 1 + mismatchNumberOfExpression(blin, ass.s.<expression>)
else mismatchNumberOfExpression(blin, ass.s.<expression>)
endif

if \( \neg \text{isSameSort}(\text{varSort}, \text{expSort}) \) then
  1 + mismatchNumberOfExpression(blin, ass.s.<expression>) +
  mismatchNumberInIndexVariable(blin, ass.s.<variable>)
else
  mismatchNumberOfExpression(blin, ass.s.<expression>) +
  mismatchNumberInIndexVariable(blin, ass.s.<variable>)
endif
endcase

mismatchNumberInIndexVariable(blin:BINDINGLIST_0, var:<indexed variable>): NAT=def
let acp=var.s.<actual parameter>-seq in
  if var.s.<variable>\in<indexed variable> then
    if \( \neg \text{isSameSort}(\text{getStaticSort}(acp[1],blin), \text{getIndexSort}(\text{getVariableSort}(var))) \) then
      1
    else
      0
    endif
  else
    mismatchNumberInIndexVariable(blin, var.s.<variable>)
  endif
endlet

mismatchNumberOfDecision(blin:BINDINGLIST_0, d: <decision>): NAT=def
let q=d.s.<question>, rcl=d.rangeConditionList_0 in
  mismatchNumberOfExpression(blin, q) + mismatchNumberOfRangeConditionList(blin, rcl)
endlet

mismatchNumberOfRangeConditionList(blin:BINDINGLIST_0, rcl: <range condition>*): NAT=def
if rcl.\=empty then empty
else
  mismatchNumberOfRangeCond(blin, rcl.head) + mismatchNumberOfRangeConditionList_0 (blin, rcl.tail)
endif

mismatchNumberOfRangeCond(blin:BINDINGLIST_0, rl: <range>*): NAT=def
if rl.\=empty then empty
else
  mismatchNumberOfRangeCond_0(blin, rl.head) + mismatchNumberOfRangeCond_0 (blin, rl.tail)
endif

mismatchNumberOfRange_0(blin:BINDINGLIST_0, range: <range>): NAT=def
 case range of
  |<closed range>=>
    mismatchNumberOfExpression(blin, range.s.<constant>) +
    mismatchNumberOfExpression(blin, range.s2.<constant>)
  |<open range>=>
    if range\in<constant> then mismatchNumberOfExpression(blin,range)
    else
      mismatchNumberOfExpression(blin,range.s.<constant>) +
      mismatchNumberOfExpression(blin,range.s2.<constant>)
    endif
endcase

mismatchNumberOfExpression_0(blin:BINDINGLIST_0, exp: <expression>): NAT=def
 case exp of
  |<create expression>=><mismatchNumberOfCreateExp_0(blin, exp)
  |<value returning procedure call>=>
    if exp\in<procedure call body> then
      mismatchNumberOfProcedureCallBody(blin, call)
    else
      mismatchNumberOfRemoteProcCallBody(blin, call)
    endif

|<range check expression> => mismatchNumberOfExpression_{bl, exp.s-<expression>} |
|<equality expression> => mismatchNumberOfExpression_{bl, exp.s-<expression>} + mismatchNumberOfExpression_{bl, exp.s2-<expression>} |
|<binary expression> => let opDef = getDefinitionInBindingList_{exp.s-implicit, bl} in |
|<conditional expression> => let opDef = getDefinitionInBindingList_{exp.s-implicit, bl} in |
|<method application> => mismatchNumberOfMethodApp |
|<operator application> => mismatchNumberOfOpApp |
|<expression gen primary> => let exp = getCreatedAgentDefinition_{bl, ce: <create expression>}: NAT=def |
|<type expression> => let exp = getCreatedAgentDefinition_{bl, ce: <create expression>}: NAT=def |
|<identifier> => let fssl = getEntityDefinition{id, procedure}, formalParameterSortList in |
|<create expression> => mismatchNumberOfParas_{bl, def, formalParameterSortList0, ce.actualParameterList0} + mismatchNumberOfActualParas_{bl, ce.actualParameterList0} |
|<remote procedure call body> => let rpd = getEntityDefinition{body, s-<identifier>, .remote procedure} in |
|<primary> => let rpd = getEntityDefinition{body, s-<identifier>, .remote procedure} in |
|<procedure call body> => mismatchNumberOfExpression_{bl, exp.s-<expression>} + mismatchNumberOfExpression_{bl, exp.s2-<expression>} |
|<alternative expression> => otherwise 0 |
mismatchNumberOfOpApp\(_{\text{bl: BINDINGLIST}\_0, \text{oa: <operator application>}}\): \text{NAT} = \text{def}

let \text{opDef} = \text{getDefinitionInBindingList}(\text{oa, s:<operation identifier>}, \text{s:<name>}, \text{bl}) in

let \text{fpsl} = \text{opDef.operationParameterSortList}\_0 \text{in}

\text{let} \text{apl} = \text{oa.actualParameterList}\_0 \text{in} 
\quad \text{mismatchNumberOfParas}\_0(\text{bl, fpsl, apl}) + \text{mismatchNumberOfActualParas}\_0(\text{bl, apl})
\text{endif}

mismatchNumberOfMethodApp\(_{\text{bl: BINDINGLIST}\_0, \text{ma: <method application>}}\): \text{NAT} = \text{def}

let \text{opDef} = \text{getDefinitionInBindingList}(\text{ma, s:<operation identifier>}, \text{s:<name>}, \text{bl}) in

let \text{fpsl} = \text{opDef.operationParameterSortList}\_0 \text{in}

\text{let} \text{apl} = \text{ma.actualParameterList}\_0 \text{in}
\quad \text{mismatchNumberOfParas}\_0(\text{bl, fpsl, apl}) + \text{mismatchNumberOfActualParas}\_0(\text{bl, apl})
\text{endif}

mismatchNumberOfParas\_0(\text{bl: BINDINGLIST}\_0, \text{fpsl: <sort>*}, \text{apl: <expression>*}): \text{NAT} = \text{def}

if \text{fpsl} = \text{empty} then 0
\quad \text{elseif} \text{fpsl.head=undefined} \text{visSameSort}(\text{fpsl.head, \text{apl.head}}) then \text{mismatchNumberOfParas}\_0(\text{bl, fpsl.tail}, \text{apl.tail})
\quad \text{else} \text{mismatchNumberOfParas}\_0(\text{bl, fpsl.tail, apl.tail})+1
\text{endif}

mismatchNumberOfActualParas\_0(\text{bl: BINDINGLIST}\_0, \text{apl: <expression>*}): \text{NAT} = \text{def}

if \text{apl} = \text{empty} then 0
\quad \text{else} \text{mismatchNumberOfExpression}\_0(\text{bl, apl.head}) + \text{mismatchNumberOfActualParas}\_0(\text{bl, apl.tail})
\text{endif}

\text{formalParameterSortList}\_0(\text{d: <agent definition}>|<agent type definition>|<composite state>|<composite state type definition>|<procedure definitions>|<remote procedure definition>): <\text{sort}>* = \text{def}

\text{case} \text{d of}
\quad |<\text{agent definition}>|<\text{agent type definition}>|<\text{composite state}>|<\text{composite state type definition}>|<\text{procedure definitions}>|<\text{remote procedure definition}>:
\quad |\text{<fp.parentASI.s:<sort> | fp in } \text{d.agentFormalParameterList}\_0\rangle
\quad |\text{<procedure definitions>}
\quad |\text{<fp.parentASI.s:<sort> | fp in } \text{d.procedureFormalParameterList}\_0\rangle
\quad |\text{<remote procedure definition>}
\quad |\text{<fp.s:<sort> | fp in } \text{d.s:<procedure signature>}, \text{s:<formal parameter>}}
\text{endcase}

The function \text{staticSortSet}\_0 gets the possible static sort set for an <expression>.

\text{staticSortSet}\_0(\text{exp: <expression>}: <\text{sort}> \text{-set} = \text{def}
\quad \{\text{getStaticSortSet}(\text{exp, bl): bl<getBindingListSet(\text{exp.contextOfExp}\_0)}\}

The function \text{contextOfExp}\_0 finds the context that the <expression> is in.

\text{contextOfExp}\_0(\text{exp: <expression>}: \text{CONTEXT}\_0 = \text{def}
\quad \text{if} (\text{class <assignment>: isAncestorASI0(ass, exp)}) \text{then}
\quad \text{parentASI0fKind}(\text{exp, <assignment>})
\quad \text{elseif (class <decision>: isAncestorASI1(ass, exp)}) \text{then}
\quad \text{parentASI0fKind}(\text{exp, <decision>})
\quad \text{elseif (exp1 in <expression>: isAncestorASI0(exp1, exp)}) \text{then}
\quad \text{parentASI0fKind}(\text{exp, <expression>}).\text{contextOfExp}\_0
\quad \text{endif}

The function \text{getPredefinedOpParas}\_0 gets the sort list of the formal parameters of a predefined operation. This function is not formally defined in this Recommendation.

\text{getPredefinedOpParas}\_0: \text{PREDEFINEDOPERATION}\_0 \to <\text{sort}>*

The function \text{getPredefinedOpResults}\_0 gets the result sort of a predefined operation. This function is not formally defined in this Recommendation.
2.2.3 Path Item

Abstract Syntax

\[
\text{Path-item} = \text{Package-qualifier} \mid \text{Agent-type-qualifier} \mid \text{Agent-qualifier} \mid \text{State-type-qualifier} \mid \text{State-qualifier} \mid \text{Data-type-qualifier} \mid \text{Procedure-qualifier} \mid \text{Signal-qualifier} \mid \text{Interface-qualifier}
\]

<table>
<thead>
<tr>
<th>Concrete Syntax</th>
</tr>
</thead>
</table>
| \[
\text{<path item> :: [<scope unit kind>] <name>}
\] |
| \[
\text{<scope unit kind> = package system type system block block type process process type state state type procedure signal type operator method interface}
\] |

Mapping to Abstract Syntax

\[
\begin{align*}
\text{<path item>}(\text{PACKAGE},n) & \Rightarrow \text{mk-Package-qualifier}(n) \\
\text{<path item>}(\text{SYSTEM},n) & \Rightarrow \text{mk-Agent-qualifier}(n) \\
\text{<path item>}(\text{SYSTEM TYPE},n) & \Rightarrow \text{mk-Agent-type-qualifier}(n) \\
\text{<path item>}(\text{BLOCK},n) & \Rightarrow \text{mk-Agent-qualifier}(n) \\
\text{<path item>}(\text{BLOCK TYPE},n) & \Rightarrow \text{mk-Agent-type-qualifier}(n) \\
\text{<path item>}(\text{PROCESS},n) & \Rightarrow \text{mk-Agent-qualifier}(n) \\
\text{<path item>}(\text{PROCESS TYPE},n) & \Rightarrow \text{mk-Agent-type-qualifier}(n) \\
\text{<path item>}(\text{STATE},n) & \Rightarrow \text{mk-State-qualifier}(n) \\
\text{<path item>}(\text{STATETYPE},n) & \Rightarrow \text{mk-State-type-qualifier}(n) \\
\text{<path item>}(\text{PROCEDURE},n) & \Rightarrow \text{mk-Procedure-qualifier}(n) \\
\text{<path item>}(\text{SIGNAL},n) & \Rightarrow \text{mk-Procedure-qualifier}(n) \\
\text{<path item>}(\text{OPERATOR},n) & \Rightarrow \text{mk-Procedure-qualifier}(n) \\
\text{<path item>}(\text{METHOD},n) & \Rightarrow \text{mk-Procedure-qualifier}(n) \\
\text{<path item>}(\text{TYPE},n) & \Rightarrow \text{mk-Data-type-qualifier}(n) \\
\text{<path item>}(\text{INTERFACE},n) & \Rightarrow \text{mk-Interface-qualifier}(n)
\end{align*}
\]
2.3 Informal Text

Abstract Syntax

\[ \text{Informal-text} :: \ldots \]

Concrete Syntax

\[ \langle \text{informal text} \rangle :: \langle \text{character string} \rangle \]

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

\[ \text{SDL-specification} :: [ \text{Agent-definition} ] \text{Package-definition-set} \]

Concrete Syntax

\[ \langle \text{sdl specification} \rangle :: [ \langle \text{textual system specification} \rangle ] \langle \text{package definition} \rangle ^* \langle \text{referenced definition} \rangle ^* \]

\[ \langle \text{textual system specification} \rangle = \]
\[ \langle \text{agent definition} \rangle \]
\[ | \langle \text{textual typebased agent definition} \rangle \]

Transformations

\[ \langle \text{sdl specification} \rangle (\text{sys}, p, r) \]
\[ \text{provided } \text{sys} \in (\langle \text{process definition} \rangle \cup \langle \text{textual typebased process definition} \rangle \cup \]
\[ \langle \text{block definition} \rangle \cup \langle \text{textual typebased block definition} \rangle ) \]
\[ =1=> \langle \text{sdl specification} \rangle (\]
\[ \langle \text{system definition} \rangle (\text{empty}, \]
\[ \langle \text{system heading} \rangle (\text{sys.name}_0, \langle \text{agent additional heading} \rangle (\text{undefined, empty})), \]
\[ \langle \text{agent structure} \rangle (\text{undefined, } \langle \text{sys} \rangle, \langle \text{agent body} \rangle (\text{undefined, empty})), \]
\[ p, r) \]

A \langle \text{system specification} \rangle being a \langle \text{process definition} \rangle or a \langle \text{textual typebased process definition} \rangle is derived syntax for a
\langle \text{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.} \rangle

A \langle \text{system specification} \rangle being a \langle \text{block definition} \rangle or a \langle \text{textual typebased block definition} \rangle is derived syntax for a
\langle \text{system definition} \rangle having the same name as the block, containing implicit channels and containing the \langle \text{block definition} \rangle or \langle \text{textual typebased block definition} \rangle as the only definition.

Mapping to Abstract Syntax

\[ | \langle \text{sdl specification} \rangle (s, \text{packages}, *) => \]
\[ \text{mk-SDL-specification}(\text{Mapping}(s), \text{Mapping}(\text{packages})) \]

2.4.2 Package

Abstract Syntax

\[ \text{Package-definition} :: \]
\[ \text{Package-name} \]
\[ \text{Package-definition-set} \]
\[ \text{Data-type-definition-set} \]
Concrete Syntax

\[
\text{<package definition>} :: \text{<package use clause>}* \text{<package heading>} \text{<entity in package>}*
\]

\[
\text{<package heading>} :: \text{<qualifier> <package <name> <package interface>}
\]

\[
\text{<entity in package>} =
\begin{align*}
\text{<agent type definition>} \\
\text{<agent type reference>} \\
\text{<package definition>} \\
\text{<package reference>} \\
\text{<signal definition>} \\
\text{<signal reference>} \\
\text{<signal list definition>} \\
\text{<remote variable definition>} \\
\text{<data definition>} \\
\text{<data type reference>} \\
\text{<procedure definitions>} \\
\text{<procedure reference>} \\
\text{<remote procedure definition>} \\
\text{<composite state type definition>} \\
\text{<composite state type reference>} \\
\text{<exception definition>} \\
\text{<select definition>} \\
\text{<interface reference>}
\end{align*}
\]

\[
\text{<package use clause>} :: \text{<package <identifier> <definition selection>}*}
\]

\[
\text{<definition selection>} :: \{<\text{selected entity kind}>\} \text{<name>}
\]

\[
\text{<selected entity kind>} =
\begin{align*}
\text{system type} \\
\text{block type} \\
\text{process type} \\
\text{package} \\
\text{signal} \\
\text{procedure} \\
\text{remote procedure} \\
\text{type} \\
\text{signallist} \\
\text{state type} \\
\text{synonym} \\
\text{remote} \\
\text{exception} \\
\text{interface}
\end{align*}
\]

\[
\text{<package interface>} = \text{<definition selection}*}
\]

Conditions on Concrete Syntax

\[
\forall id \in <\text{identifier}>: \\
(id.\text{parentAS0} \in \text{<package use clause>}) \Rightarrow \text{getEntityDefinition}(id, \text{package}) \neq \text{undefined}
\]
For each `<package<identifier>` mentioned in a `<package use clause>`, there must exist a corresponding visible `<package>`.

\[ \forall \text{pd} \in \text{<package definition>} : \text{pd}.\text{parentAS0} \in \text{<sdl specification>} \Rightarrow \text{pd}.\text{qualifier}_0 = \text{undefined} \]

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

\[ \forall \text{ds1, ds2} \in \text{<definition selection>} : (\text{ds1}.\text{parentAS0} = \text{ds2}.\text{parentAS0} \wedge \text{ds1} \neq \text{ds2}) \Rightarrow (\forall \text{ds1.s} <\text{selected entity kind}> = \text{ds2.s}<\text{name}> \vee \text{ds1.s}<\text{name}> \neq \text{ds2.s}<\text{name}>) \]

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

\[ \forall \text{ds} \in \text{<definition selection>} : \text{ds}.\text{s}<\text{selected entity kind}> = \text{undefined} \wedge \text{d}.\text{parentAS0} \in \text{<package interface>} \Rightarrow (\exists \text{d} \in \text{ENTITYDEFINITION}_0 \cup \text{REFERENCE}_0 : \text{d}.\text{surroundingScopeUnit}_0 = \text{ds}.\text{parentAS0} \wedge \text{d}.\text{entityName}_0 = \text{ds.s}<\text{name}>) \]

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

\[ \forall \text{uc} \in \text{<package use clause>} : \forall \text{ds} \in \text{<definition selection>} : \text{let pd = uc.usedPackage0 in ds}.\text{s}<\text{selected entity kind}> = \text{undefined} \wedge \text{d}.\text{parentAS0} \in \text{<package interface>} \Rightarrow (\exists \text{d} \in \text{ENTITYDEFINITION}_0 \cup \text{REFERENCE}_0 : \text{d}.\text{surroundingScopeUnit}_0 = \text{ds}.\text{parentAS0} \wedge \text{d}.\text{entityName}_0 = \text{ds.s}<\text{name}> \wedge \text{d}.\text{entityKind}_0 = \text{ds.s}<\text{selected entity kind}> ) \]

For a `<definition selection>` in a `<package use clause>`, the `<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.

\[ \forall \text{ds} \in \text{<definition selection>} : \text{ds}.\text{s}<\text{selected entity kind}> \neq \text{undefined} \wedge \text{d}.\text{parentAS0} \in \text{<package interface>} \Rightarrow (\exists \text{d} \in \text{ENTITYDEFINITION}_0 \cup \text{REFERENCE}_0 : \text{d}.\text{surroundingScopeUnit}_0 = \text{ds}.\text{surroundingScopeUnit}_0 \wedge \text{d}.\text{entityName}_0 = \text{ds.s}<\text{name}> \wedge \text{d}.\text{entityKind}_0 = \text{ds.s}<\text{selected entity kind}> ) \]

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

\[ \text{let usePredef = <package use clause>(<identifier>(<identifier>(<identifier>("Predefined"), undefined, <name>("Predefined")), undefined) in <package definition>(uses, heading, entities) provided uses.head ≠ usePredef =6=> <package definition>(< usePredef > ^ uses, heading, entities) } \]

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

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

\[ \text{use Predefined; } \]

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\textsubscript{0} gets the used package definition list of a scope unit.

\begin{align*}
\text{usedPackageDefinitionList}_0(\text{su: SCOPEUNIT}_0): \text{<package definition>} & \overset{\ast}{=}_\text{def}
<u\text{.usedPackage}_0 \mid u \text{ in } su.s->\text{<package use clause}>}
\end{align*}

The function usedPackage\textsubscript{0} gets the package definition for a <package use clause>.

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

2.4.3 Referenced definition

Concrete Syntax

<referenced definition> =
<definition>

<definition> =
\begin{align*}
| & <package definition> \\
| & <agent definition> \\
| & <agent type definition> \\
| & <composite state> \\
| & <composite state type definition> \\
| & <procedure definitions> \\
| & <operation definition>
\end{align*}

<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

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

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

∀ref ∈ REFERENCE0 : ref.referencedDefinition0 ≠ undefined

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

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

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

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

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

∀def ∈ ENTITYDEFINITION0 : def ∈ <referenced definition> ⇒ def.qualifier0 = undefined

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

Transformations

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

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

b = <block reference>(*, inst) =4=> adaptDefinition(b.referencedDefinition0, inst)
p = <process reference>(* ,* ) = > adaptDefinition(p.referencedDefinition0, inst)
i= <interface reference>(* ,* ) = > adaptDefinition(i.referencedDefinition0, undefined)
c= <composite state reference>(* ) = > adaptDefinition(c.referencedDefinition0, undefined)
s= <system type reference>(* ) = > adaptDefinition(s.referencedDefinition0, undefined)
b= <block type reference>(* ,* ) = > adaptDefinition(b.referencedDefinition0, undefined)
p= <process type reference>(* ,* ) = > adaptDefinition(p.referencedDefinition0, undefined)
c= <composite state type reference>(* ,* ) = > adaptDefinition(c.referencedDefinition0, undefined)

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

Auxiliary Functions

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

referencedDefinition0(ref: REFERENCE0): <referenced definition> = def
let eKind = ref.referencedKind0 in
let refName = ref.referencedName0 in
if (∃! d ∈ <referenced definition>: isAncestorAS0(d.parentAS0, ref) ∧ d.entityName0 = refName ∧ d.entityKind0 = eKind) then
  let d = take({d ∈ <referenced definition>: isAncestorAS0(d.parentAS0, ref) ∧ d.entityName0 = refName ∧ d.entityKind0 = eKind}) in
  if isQualifierMatched0(d.qualifier0, ref.surroundingScopeUnit0) then d
else
  undefined
endif
endif
endlet

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

referencedBy0(rd: <referenced definition>): REFERENCE0 = def
take({ref ∈ REFERENCE0: ref.referencedDefinition0 = rd})

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

isPathItemMatched0(seq1: <path item>* , seq2: <path item>* ): BOOLEAN = def
(seq1.length ≤ seq2.length ∧
(∀i ∈ 1..seq1.length:
  seq1[i].s<s-name> = seq2[i].s<s-name> ∧ seq1[i].s<s-scope unit kind> = seq2[i].s<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,
2.4.4 Select Definition

Concrete Syntax

```plaintext
<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 definition> |
   <composite type definition> |
   <composite state type definition> |
   <state partition> |
   <timer definition> |
   <variable definition> |
   <procedure definitions> |
   <procedure reference>
```

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

```plaintext
mergeNumbers(inst1: <number of instances>, inst2: <number of instances>): <number of instances> =def
if inst1 = undefined then inst2
else if 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
```

otherwise undefined
endcase

Transformations

\[
\begin{align*}
\text{<select definition>}(\text{cond}, \text{defs}) & \Rightarrow \begin{cases} 
\text{if value}(\text{cond}) \text{ then defs} & \text{else empty} \end{cases} \text{ endif}
\end{align*}
\]

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

\[
\text{<transition option>}::=\text{<alternative question>} \text{ <decision body>}
\]

\[
\text{<alternative question>} = \text{<simple expression>} | \text{<informal text>}
\]

Conditions on Concrete Syntax

\[
\forall \text{ce} \in \text{<constant expression>}:\text{parentAS0ofKind}(\text{ce}, \text{<transition option>}) \neq \text{undefined} \Rightarrow \text{isSimpleExpression}(\text{ce})
\]

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

\[
\forall \text{toe} \in \text{<transition option>}::\exists \text{<simple expression>} \in \text{<alternative question>}\Rightarrow \\
(\forall \text{rc} \in \text{<range condition>>\text{rc in to.e}<decision body>}::\exists \text{answerPart}\Rightarrow \\
(\forall \text{ce} \in \text{<constant expression>: isAncestorAS0}(\text{rc}, \text{ce}) \Rightarrow \\
\text{isSameSort}(\text{to.e}<alternative question>.\text{staticSort, ce.staticSort}))
\]

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

\[
\begin{align*}
t &= \text{<transition gen action statement>}(a, \text{undefined}) \quad \text{provided } a.\text{last} \notin \text{<transition option>} \land t.\text{parentAS0}.\text{parentAS0} \in \text{<transition option>} \land \\
&\quad t.\text{findContinueLabel} \neq \text{undefined} \Rightarrow \text{<transition gen action statement>}(a, \\
&\quad \text{<terminator statement>}(\text{undefined, <join>}(\text{findContinueLabel}(t))))
\end{align*}
\]

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.

\[
\begin{align*}
t &= \text{<transition option>}(q, \text{<decision body>}(\text{answers, elsePart})) \Rightarrow \begin{cases} 
\text{let matching} = \{ a.s<transition> | a \in \text{answers.toSet}: q.\text{value} \in a.s<answer> \} \text{ in} \\
\text{if matching} \neq \emptyset \text{ then matching.take} & \text{elseif elsePart} \neq \text{undefined} \text{ then elsePart.s}<transition> \\
\text{else undefined} & \text{endif} \\
\text{endlet}
\end{cases}
\end{align*}
\]

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

\[\forall d \in \text{Agent-type-definition}: d.\text{agentKind}, \in \{\text{system, process}\} \Rightarrow (d.s.-\text{Agent-definition-set} \neq \emptyset \lor d.s.-\text{State-machine-definition} \neq \text{undefined})\]

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.

\[\forall d \in \text{Agent-type-definition}: (d.\text{agentKind}, = \text{process}) \Rightarrow (\forall d' \in \text{Agent-type-definition} \cup \text{Agent-definition}: d'.\text{parentAS0} = d \Rightarrow d'.\text{agentKind}, = \text{process})\]

The contained Agent-definitions and Agent-type-definitions 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.name0, 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(d: <agent type definition>):<procedure definition>-set =
{ pd<procedure definition>: pd.surroundingScopeUnit0 = d }
```

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

```
validOutputSignalSet(d: <agent type definition> ∪ <agent definition>): SIGNAL0 =
d.localOutputSignalSet0 ∪ d.inheritedOutputSignalSet0
```

```
localOutputSignalSet(d: <agent type definition> ∪ <agent definition>): SIGNAL0 =
{ sid < SIGNAL0:
(∃ gate constraint:
 (gc.parentAS0 < < textual gate definition>) ∧
isDefinedIn gc.parentAS0, d ∧
(gc.direction0 = out) ∧
(sid.signalSet0(gc.s-<signal list item>-seq))) ∨
(∃ path:
 (cp.parentAS0 < < channel definition>) ∧
isDefinedIn cp.parentAS0, d ∧
(cp.destination0.s-env#undefined) ∧
(sid.signalSet0(cp.s-<signal list item>-seq)))
}
```

```
inheritedOutputSignalSet(d: <agent type definition> ∪ <agent definition>): SIGNAL0 =
if d.specialization0 = undefined then ∅
else d.specialization0.s-<type expression>.baseType0.validOutputSignalSet0
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

```
∀ fcpe <formal context parameter>:
 (fcpe.surroundingScopeUnit0 < < system type definition> ) ⇒
 (fcpe < 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.surroundingScopeUnit0 < < 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))),
```

Concrete Syntax

2.5.1.4 Process Type

Concrete Syntax

\[
\text{Concrete Syntax}
\]

\[
\text{2.5.1.3 Block Type}
\]

Mapping to Abstract Syntax

\[
\text{Mapping to Abstract Syntax}
\]

2.5.1.4 Process Type

Concrete Syntax

\[
\text{Concrete Syntax}
\]

\[
\text{Mapping to Abstract Syntax}
\]
2.5.1.5 Composite State Type

Abstract Syntax

\[
\text{Composite-state-type-definition} :: \begin{align*}
& \text{State-type-name} \\
& \text{[ Composite-state-type-identifier ]} \\
& \text{Composite-state-formal-parameter*} \\
& \text{State-entry-point-definition-set} \\
& \text{State-exit-point-definition-set} \\
& \text{Gate-definition-set} \\
& \text{Data-type-definition-set} \\
& \text{Syntype-definition-set} \\
& \text{Exception-definition-set} \\
& \text{Composite-state-type-definition-set} \\
& \text{Variable-definition-set} \\
& \text{Procedure-definition-set} \\
& \text{[ Composite-state-graph | State-aggregation-node ]}
\end{align*}
\]

Conditions on Abstract Syntax

\[\forall d \in \text{Composite-state-type-definition}: \text{d.s-Gate-definition-set} \neq \emptyset \Rightarrow \exists s \in \text{State-machine-definition}: \text{getEntityDefinition}(s.d.s-\text{Composite-state-type-identifier}, \text{state type}) = d\]

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

Concrete Syntax

\[
\text{<composite state type definition>} :: \begin{align*}
& \text{<package use clause>*} \text{ <composite state type heading>} \text{ <composite state structure>}
\end{align*}
\]

Mapping to Abstract Syntax

\[
\text{-> \text{mk-Composite-state-type-definition}(Mapping(name), Mapping(parent), Mapping(params),} \\
\text{BigSeq(Mapping(<c in conns: (c \in <state entry points>) >)),} \\
\text{BigSeq(Mapping(<c in conns: (c \in <state exit points>) >)),} \\
\text{Mapping(gates),} \\
\text{\{ e \in Mapping(entities).toSet: (e \in Data-type-definition)\},} \\
\text{\{ e \in Mapping(entities).toSet: (e \in Syntype-definition)\},}
\]
2.5.2 Type Expression

Concrete Syntax

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

Conditions on Concrete Syntax

\( \forall t \in <\text{type expression}> \): 
\( t . s . <\text{actual context parameter} > - \text{seq} \neq \text{empty} \Rightarrow \text{isParameterizedType}(t . e . \text{baseType}_0) \)

<actual context parameters> can be specified if and only if <base type> denotes a parameterized type.

Transformations

let \( mn = \text{name} \) in
\( t = <\text{type expression}> ((id = <\text{identifier}>(q, n), \text{params}) \)
provided \( \text{params} \neq \text{undefined} \wedge \text{params} \neq \text{empty} \wedge t . \text{parentAS0} \neq \text{specialization} \)
\( =11 \Rightarrow <\text{type expression}> ((<\text{identifier}>(q, mn), \text{undefined}) \)
and
\( <\text{id refersto}_0 \) > 
\( \Rightarrow <\text{id refersto}_0 \)
replaceContextParameters( id refersto_0 , localFormalContextParameterList_0 , params , 
createNewType( 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>. 

ITU-T Z.100/Annex F2 (11/2000) 45
Mapping to Abstract Syntax

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

Auxiliary Functions

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

\[ \text{isDirectSubType}(ed: \text{ENTITYDEFINITION}_0, td: \text{TYPEDEFINITION}_0): \text{BOOLEAN} = \exists te \in <\text{type expression}>: \text{te.parentAS} = ed.\text{specialization} \land td = \text{te.baseType} \]

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

\[ \text{isSubtype}(sub: \text{ENTITYDEFINITION}_0, sup: \text{TYPEDEFINITION}_0): \text{BOOLEAN} = \text{isDirectSubType}(sub, sup) \lor (\exists ttd \in \text{TYPEDEFINITION}_0: \text{isSubtype}(sub, ttd) \land \text{isSubtype}(ttd, sup)) \]

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

\[ \text{baseType}(te: <\text{type expression}>): \text{TYPEDEFINITION}_0 = \text{getEntityDefinition}(te.\text{s}<\text{base type}, te.\text{baseTypeKind}) \]

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

\[ \text{baseTypeKind}(te: <\text{type expression}>): \text{ENTITYKIND}_0 = \begin{cases} \text{agent type} & \text{if } (te.\text{surroundingScopeUnit} \in <\text{agent definition} ) \\
\text{state type} & \text{otherwise} \end{cases} \]

The function isParameterizedType determines if a type definition is a parameterized type.

\[ \text{isParameterizedType}(td: \text{TYPEDEFINITION}_0): \text{BOOLEAN} = (td.\text{formalContextParameterList} = \text{empty}) \]

Get the formal context parameter list of a type definition.

\[ \text{formalContextParameterList}(td: \text{TYPEDEFINITION}_0): <\text{name}>* = td.\text{inheritedFormalContextParameterList} \lor td.\text{localFormalContextParameterList} \]

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

\[ \text{inheritedFormalContextParameterList}(td: \text{TYPEDEFINITION}_0): <\text{name}>* = \begin{cases} \text{empty} & \text{if } \text{sp} = \text{undefined} \\
\text{let } \text{sp} = \text{td.specialization} \text{ in} \\
\text{case } \text{sp} \text{ of} \\
\text{<interface specialization} => <\text{getUnboundFormalContextParameterList}(tel.\text{baseType}, \text{formalContextParameterList}_0, tel.\text{actualContextParameterList}_0) \\
\text{<procedure call body} => <\text{procedure} \\
\text{<typebased system heading} => <\text{system type} \\
\text{<typebased block heading} => <\text{block type} \\
\text{<typebased process heading} => <\text{process type} \\
\text{<typebased composite state} => <\text{typebased state partition heading} => <\text{state type} \\
\text{otherwise} \text{undefined} \\
\end{cases} \]

\[ \text{endlet} \]
Get the unbound formal context parameter list of a formal context parameter list according to an actual context parameter list.

\[
\text{getUnboundFormalContextParameterList}(\text{fcpl}: \text{<name>}, \text{acpl}: \text{<identifier>}) :=
\begin{align*}
\text{if} & \ (\text{fcpl} = \text{empty}) \ \text{then} \ \text{empty} \\
\text{elseif} & \ (\text{acpl}.\text{head} = \text{undefined}) \ \text{then} \\
& \ <\text{fcpl}.\text{head}> \ \text{getUnboundFormalContextParameterList}(\text{fcpl}.\text{tail}, \text{acpl}.\text{tail}) \\
\text{else} & \ \text{getUnboundFormalContextParameterList}(\text{fcpl}.\text{tail}, \text{acpl}.\text{tail}) \\
\end{align*}
\]

Insert the original context parameter for the unbound ones.

\[
\text{completeFormalContextParameter}(\text{fcpl}: \text{<name>}, \text{acpl}: \text{<identifier>}) :=
\begin{align*}
\text{if} & \ (\text{fcpl} = \text{empty}) \ \text{then} \ \text{empty} \\
\text{elseif} & \ (\text{acpl}.\text{head} = \text{undefined}) \ \text{then} \\
& \ <\text{fcpl}.\text{head}> \ \text{completeFormalContextParameter}(\text{fcpl}.\text{tail}, \text{acpl}.\text{tail}) \\
\text{else} & \ <\text{acpl}.\text{head}> \ \text{completeFormalContextParameter}(\text{fcpl}.\text{tail}, \text{acpl}.\text{tail}) \\
\end{align*}
\]

Get the actual context parameter list of a type expression.

\[
\text{actualContextParameterList}(\text{te}: \text{<type expression>}) :=
\text{te}.\text{s}-\text{<actual context parameter}>-\text{seq}
\]

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

\[
\text{localFormalContextParameterList}(\text{td}: \text{TYPEDEFINITION}) :=
\begin{align*}
\text{let} & \ \text{fcps} = \text{take}(\{\text{fcps} \in \text{<agent type additional heading>} \cup \text{<composite state type heading>} \cup \\
& \ \text{<procedure heading>} \cup \text{<signal definition item>} \cup \text{<data type heading>} \cup \\
& \ \text{<interface heading>}: \text{fcps}.\text{surroundingScopeUnit}\_0 = \text{td}\} \ \text{in} \\
& \ <\text{fcpl}.\text{formalContextParameterSublist}\_0 | \text{fcpl} \in \text{fcps}.\text{s}-\text{<formal context parameter}>-\text{seq} > \\
\end{align*}
\]

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

\[
\text{formalContextParameterSublist}(\text{fcp}: \text{<formal context parameter>}) :=
\begin{align*}
\text{case} & \ \text{fcp} \ \text{of} \\
\text{| agent type context parameter} & \cup \text{| agent context parameter} => <\text{fcp}.\text{s}-\text{<name>}> \\
\text{| procedure context parameter} & \cup \text{| remote procedure context parameter} => <\text{fcp}.\text{s}-\text{<name>}> \\
\text{| signal context parameter} & => <\text{scpl}.\text{s}-\text{<name>} | \text{scpl} \in \text{fcp}.\text{s}-\text{<signal context parameter gen name}>-\text{seq} > \\
\text{| variable context parameter} & => <\text{vcpl}.\text{s}-\text{<name>} | \text{scpl} \in \text{fcp}.\text{s}-\text{<variable context parameter gen name}>-\text{seq} > \\
\text{| remote variable context parameter} & => <\text{vcpl}.\text{s}-\text{<name>} | \text{scpl} \in \text{fcp}.\text{s}-\text{<remote variable context parameter gen name}>-\text{seq} > \\
\text{| timer context parameter} & => <\text{vcpl}.\text{s}-\text{<name>} | \text{scpl} \in \text{fcp}.\text{s}-\text{<timer context parameter gen name}>-\text{seq} > \\
\text{| synonym context parameter} & => <\text{vcpl}.\text{s}-\text{<name>} | \text{scpl} \in \text{fcp}.\text{s}-\text{<synonym context parameter}>-\text{seq} > \\
\text{| sort context parameter} & => <\text{fcp}.\text{s}-\text{<name>}> \\
\text{| exception context parameter} & => <\text{vcpl}.\text{s}-\text{<name>} | \text{scpl} \in \text{fcp}.\text{s}-\text{<exception context parameter gen name}>-\text{seq} > \\
\text{| composite state type context parameter} & => <\text{fcp}.\text{s}-\text{<name>}> \\
\text{| gate context parameter} & => <\text{fcp}.\text{s}-\text{<gate>}.\text{s}-\text{<name}> >
\end{align*}
\]
Replace the context parameters by their values.

```
replaceContextParameters(p: DefinitionAS0*, v: DefinitionAS0*, orig: DefinitionAS0): DefinitionAS0 =
def
if p = empty then orig
exth else replaceContextParameters(p.tail, v.tail, replaceInSyntaxTree(p.head, v.head, orig))
endif
```

Create a new type without any formal context parameters.

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

\[ \forall ad \in \langle \text{textual typebased agent definition} \rangle : \forall te \in \langle \text{type expression} \rangle : \\
(te.parentAS0.parentAS0= ad) \Rightarrow \\
(\exists s \in \langle \text{start} \rangle : (s \in te.baseType0.startSet0) \land (s.s-<\text{name}> = \text{undefined})) \]

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

\[
\forall pd \in <procedure definitions>: \text{isAbstractType}(pd) \Rightarrow \\
\neg(\exists pc \in <procedure call>: pd = pc.\text{calledProcedure}_0)
\]

An abstract procedure cannot be called.

\[
\forall ad \in <textual typebased agent definition>: \forall te \in <type expression>: \\
te.\text{parentAS}0.\text{parentAS}0 = ad \Rightarrow \neg\text{isAbstractType}(te.\text{baseType}_0)
\]

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

\[
\forall td \in \text{TYPEDEFINITION}_0: \text{isAbstractType}(td) \Rightarrow \\
\neg(\exists d \in <textual typebased agent definition> \cup <textual typebased state partition definition> \cup <typebased composite state>: \exists te \in <type expression>: \\
(te.\text{parentAS}0.\text{parentAS}0 = d) \land (te.\text{parentAS}0 = d)) \land (te.\text{baseType}_0 = td)
\]

An abstract type can not be instantiated.

Auxiliary Functions

Determine if a type definition is abstract.

\[
\text{isAbstractType}(td: \text{TYPEDEFINITION}_0): \text{BOOLEAN} = \exists ab \in <abstract>: ab.\text{surroundingScopeUnit}_0 = td
\]

Get the \(<procedure definition>\) denoted by a \(<procedure call>\).

\[
\text{calledProcedure}_0(pc: <procedure call> \cup <value returning procedure call>): <procedure definition> = \begin{cases} 
\text{case pc of} \\
| <procedure call> => & \begin{cases} 
\text{let } t = pc.s-<procedure call body>: s-\text{implicit in} \\
\text{if } \_ \text{te } <\text{identifier} \text{ then getEntityDefinition}_0(t, \text{procedure}) \\
\text{else } t.\text{baseType}_0 \text{ // } \_ \text{te } <\text{type expression}>
\end{cases} \\
| <procedure call body> => & \begin{cases} 
\text{let } t = pc.s-<procedure call body>: s-\text{implicit in} \\
\text{if } \_ \text{te } <\text{identifier} \text{ then getEntityDefinition}_0(t, \text{procedure}) \\
\text{else } t.\text{baseType}_0 \text{ // } \_ \text{te } <\text{type expression}>
\end{cases} \\
| \text{otherwise } = & \text{undefined}
\end{cases}
\end{cases}
\]

2.5.5 Type Reference

Type references do not have a semantics in SDL.

2.5.6 Gate

Abstract Syntax

\[
\begin{align*}
\text{Gate-definition} & :: \text{Gate-name} \\
& \text{In-signal-identifier-set} \\
& \text{Out-signal-identifier-set}
\end{align*}
\]

\[
\begin{align*}
\text{In-signal-identifier} & = \text{Signal-identifier} \\
\text{Out-signal-identifier} & = \text{Signal-identifier}
\end{align*}
\]
Concrete Syntax

\[
<\text{gate in definition}> =<\text{textual gate definition}> | <\text{textual interface gate definition}>
\]

\[
<\text{textual gate definition}> ::=<\text{gate}> <\text{gate constraint}> [<\text{gate constraint}>]
\]

\[
<\text{textual interface gate definition}> ::=\{\text{ out | in }\} <\text{identifier}>
\]

\[
<\text{gate}> =<\text{name}>
\]

\[
<\text{gate constraint}> ::=
\{/\text{ out | in }\} \{<\text{textual endpoint constraint}>\}\text{ signal list item}^* \]

Conditions on Concrete Syntax

\[
\forall \text{te} \in <\text{type expression}>:\ (\text{te}.parentAS0_0 \in <\text{typebased composite state}) \lor \ (\text{te}.parentAS0_0.\text{parentAS0}_0 <\text{textual typebased agent definition}>) \lor \ (\text{te}.parentAS0_0 <\text{textual typebased state partition definition}>) \Rightarrow (\forall \text{gc} \in <\text{gate constraint}> :\ (\text{gc}.parentAS0_0 <\text{textual gate definition}>) \land \text{isDefinedIn}_0(\text{gc}.parentAS0_0, \text{te}.\text{baseType}_0) \Rightarrow \text{gc}.s-<\text{signal list item}>-\text{seq} \neq \text{empty})
\]

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

\[
\forall \text{gd} \in <\text{textual gate definition}>: \forall \text{gc} \in <\text{gate constraint}>:\ (\text{gc}.parentAS0_0 = \text{gd}) \Rightarrow (\text{let td} = \text{gd}.\text{surroundingScopeUnit}_0 \text{ in} (\text{td}.\text{entityKind}_0 = \text{state type}) \Rightarrow (\exists \text{td'} \in \text{TYPEDEFINITION}_0:\text{td'} = \text{getEntityDefinition}(\text{gc}.s-<\text{textual endpoint constraint}>, \text{state type})) \land (\text{isConsistentKindTo}_0(\text{td}.\text{entityKind}_0, \text{agent type})) \Rightarrow (\exists \text{td'} \in \text{TYPEDEFINITION}_0:\text{td'} = \text{getEntityDefinition}(\text{gc}.s-<\text{textual endpoint constraint}>, \text{agent type}))) \land (\text{endlet})
\]

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.

\[
\forall \text{gc} \in <\text{gate constraint}>:\forall \text{ce} \in <\text{channel endpoint}>:\forall \text{ce'} \in <\text{channel endpoint}>:\ (\text{gc}.parentAS0_0.s-<\text{gate}> = \text{ce}.s-<\text{gate}> ) \land (\text{ce} \neq \text{ce'}) \land (\text{ce}.parentAS0_0 = \text{ce'}.parentAS0_0 ) \land (\text{ce}.parentAS0_0 <\text{channel path}>) \land (\text{gc}.parentAS0_0 <\text{textual gate definition}>) \Rightarrow (\text{let td} = \text{getEntityDefinition}(\text{gc}.s-<\text{textual endpoint constraint}>, \text{gc}.surroundingScopeUnit0_0.\text{entityKind}_0) \text{ in} \exists \text{td'} \in \text{ENTITYDEFINITION}_0:\ (\text{td'} = \text{ce'}.\text{channelEndpointReferTo}_0) \land ((\text{td} = \text{td'}) \lor \text{isSubtype}_0(\text{td'}, \text{td})) \text{ endlet}) \land (\text{ce}.parentAS0_0.s-<\text{signal list item}>-\text{seq.signalSet}_0 \subseteq \text{gc}.s-<\text{signal list item}>-\text{seq.signalSet}_0)
\]

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 \text{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 in
td.channelDefinitionSet0 ≠ ∅) ⇒

∀ gc ∈ <gate constraint> ∀ sig ∈ SIGNAL:

gc.parentAS0 ∈ td.channelDefinitionSet0 ∧ (sig ∈ gc.s <signal list item> = seq.signalSet0) ⇒

(∃ pe ∈ <channel path>: (cp.parentAS0 ∈ td.channelDefinitionSet0) ∧

((gc.direction0 = in) ⇒

(cp.origination0.s.<gate> = gc.parentAS0.s.<gate>) ∧

((cp.origination0.s.<gate> = gc.parentAS0.s.<gate>) ∧

cp.destination0.s.<gate> = gc.parentAS0.s.<gate>) ∧

((cp.destination0.s.<gate> = gc.parentAS0.s.<gate>) ∧

(sig ∈ cp.s <signal list item> = seq.signalSet0)))) endlet)

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

∀ gc, gc' ∈ <gate constraint>:

(gc ≠ gc') ∧ (gc.parentAS0 = gc'.parentAS0) ⇒

(gc.s <textual endpoint constraint> = gc'.s <textual endpoint constraint>) ∧

((gc.direction0 = out) = (gc'.direction0 = in) ∨ (gc'.direction0 = in) ∧ (gc.direction0 = out))

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

∀ gd ∈ <textual gate definition>: (gd.s.adding ≠ undefined) ⇒

(let td = gd.s окружающийScopeUnit0 in

∃ id ∈ TYPEDEFINITION0: ∃ gd ∈ <textual gate definition>:

isSubtype(id, td') ∧ (gd ∈ td'.gateDefinitionSet0) ∧

(gd.s.<gate> = gd.s.<gate>). endlet)

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

∀ ec, ec' ∈ <textual endpoint constraint>:

isSubtype(ec.s окружающийScopeUnit0, ec'. окружающийScopeUnit0) ∧

(ec.parentAS0 <gate constraint> = ec'.parentAS0 <gate constraint>) ∧

(ec.parentAS0.parentAS0 <textual gate definition>) ∧

(ec'.parentAS0.parentAS0 <textual gate definition>) ∧

(ec.parentAS0.parentAS0.s.adding ≠ undefined) ∧

((ec.direction0 = out) ∧ (ec'.direction0 = in)) ∨ ((ec.direction0 = in) ∧ (ec'.direction0 = out)) ∧

(ec.parentAS0.parentAS0.s.<gate> = ec'.parentAS0.parentAS0.s.<gate>) ⇒

(let td = ec.s.getFeatureDefinition(ec.s.<identifier>, ec.s. окружаенныйScopeUnit0, ec.entityKind0) in

let td' = ec'.s.getFeatureDefinition(ec'.s.<identifier>, ec'.s. окружаенныйScopeUnit0, ec'.entityKind0) in

(td = td') ∨ isSubtype(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>: ∃ id ∈ <interface definition>:

id = 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

\[
t = \langle \text{textual interface gate definition} \rangle(\text{out}, id = \langle \text{identifier} \rangle(q, n)) = 1 \Rightarrow
\langle \text{textual gate definition} \rangle(n, \text{undefined},
\langle \text{gate constraint} \rangle(\text{out}, \langle \text{textual endpoint constraint} \rangle(\text{undefined}, id), \text{empty}())
\]

\[
t = \langle \text{textual interface gate definition} \rangle(\text{in}, id = \langle \text{identifier} \rangle(q, n)) = 1 \Rightarrow
\langle \text{textual gate definition} \rangle(n, \text{undefined},
\langle \text{gate constraint} \rangle(\text{in}, \langle \text{textual endpoint constraint} \rangle(\text{undefined}, id), \text{empty}())
\]

\langle \text{textual interface gate definition} \rangle \text{ and } \langle \text{graphical interface gate definition} \rangle \text{ are shorthand for a } \langle \text{textual gate definition} \rangle \text{ or a } \langle \text{graphical gate definition} \rangle, \text{ respectively, having the name of the interface as } \langle \text{gate name} \rangle \text{ and the } \langle \text{interface identifier} \rangle \text{ as the } \langle \text{gate constraint} \rangle \text{ or } \langle \text{signal list area} \rangle.

Mapping to Abstract Syntax

| \langle \text{textual gate definition} \rangle(name,
\langle \text{gate constraint} \rangle(\text{in}, *, \text{inlist}),
\langle \text{gate constraint} \rangle(\text{out}, *, \text{outlist}))
\Rightarrow \text{mk-Gate-definition}(\text{Mapping}(name), \text{Mapping(inlist)}.\text{toSet}, \text{Mapping(outlist)}.\text{toSet})

| \langle \text{textual gate definition} \rangle(name,
\langle \text{gate constraint} \rangle(\text{out}, *, \text{outlist}),
\langle \text{gate constraint} \rangle(\text{in}, *, \text{inlist}))
\Rightarrow \text{mk-Gate-definition}(\text{Mapping}(name), \text{Mapping(inlist)}.\text{toSet}, \text{Mapping(outlist)}.\text{toSet})

| \langle \text{textual gate definition} \rangle(name, \langle \text{gate constraint} \rangle(\text{in}, *, \text{inlist}), \text{undefined})
\Rightarrow \text{mk-Gate-definition}(\text{Mapping}(name), \text{Mapping(inlist)}.\text{toSet}, \emptyset)

| \langle \text{textual gate definition} \rangle(name, \langle \text{gate constraint} \rangle(\text{out}, *, \text{outlist}), \text{undefined})
\Rightarrow \text{mk-Gate-definition}(\text{Mapping}(name), \emptyset, \text{Mapping(outlist)}.\text{toSet})

Auxiliary Functions

Get the \langle \text{gate in definition} \rangle defined in an \langle \text{agent type definition} \rangle, a \langle \text{composite state type definition} \rangle, a \langle \text{agent definition} \rangle or a \langle \text{composite state} \rangle.

\text{gateDefinitionSet}(td: \langle \text{agent type definition} \rangle \cup \langle \text{composite state type definition} \rangle \cup
\langle \text{agent definition} \rangle \cup \langle \text{composite state} \rangle): \langle \text{gate in definition} \rangle - \text{set}
\text{td.localGateDefinitionSet} \cup \text{td.inheritedGateDefinitionSet}_0

\text{localGateDefinitionSet}(td: \langle \text{agent type definition} \rangle \cup \langle \text{composite state type definition} \rangle \cup
\langle \text{agent definition} \rangle \cup \langle \text{composite state} \rangle): \langle \text{gate in definition} \rangle - \text{set}
\{gd\in \langle \text{gate in definition} \rangle : gd.\text{surroundingScopeUnit}_0 = td\}

\text{inheritedGateDefinitionSet}(td: \langle \text{agent type definition} \rangle \cup \langle \text{composite state type definition} \rangle \cup
\langle \text{agent definition} \rangle \cup \langle \text{composite state} \rangle): \langle \text{gate in definition} \rangle - \text{set}
\text{let sp = td.specialization} \text{ in}
\text{if sp = undefined then } \emptyset
\text{ else sp.s.<type expression>.baseType}_0.\text{gateDefinitionSet}_0
\text{endif}
\text{endlet}

Get the \langle \text{channel definition} \rangle defined in an \langle \text{agent type definition} \rangle or a \langle \text{agent definition} \rangle.

\text{channelDefinitionSet}(td: \langle \text{agent type definition} \rangle \cup \langle \text{agent definition} \rangle): \langle \text{channel definition} \rangle - \text{set}
\text{td.localChannelDefinitionSet} \cup \text{td.inheritedChannelDefinitionSet}_0

\text{localChannelDefinitionSet}(td: \langle \text{agent type definition} \rangle \cup \langle \text{agent definition} \rangle): \langle \text{channel definition} \rangle - \text{set}
\{cd\in \langle \text{channel definition} \rangle : cd.\text{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(sl.head, interface) in
            fd.usedSignalSet_{0}∪{sd.identifier: sd.fd definesSignalSet_{0}}∪sl.tail.signalSet_{0}
        | {signallist} =>
            (let sld = getEntityDefinition(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>
    | <variable context parameter>
    | <signal context parameter>
    | <signal list item>*seq
    | <interface context parameter>
    | <sort context parameter>
    | <exception context parameter>
    | <gate context parameter>
    | <interface context parameter>

Conditions on Concrete Syntax

∀fcp∈ FORMALCONTEXTPARAMETER_{0} ∀acp∈<actual context parameter>: isContextParameterCorresponded(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}∉FORMALCONTEXTPARAMETERS) ∧
(∀fcp∈ FORMALCONTEXTPARAMETERS: fcp.contextParameterAtleastDefinition_{0}∉FORMALCONTEXTPARAMETERS)

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(fcp, acp) ⇒
    (∀td∈TYPEDEFINITION_{0}: (td = fcp.contextParameterAtleastDefinition_{0}) ⇒
\((td' \in \text{FORMALCONTEXTPARAMETER}_0) \land \neg(isParameterizedType_0(td')) \land
(3 \in \text{TYPEDEFINITION}_0; (td = \text{getEntityDefinition}_0(acp, td'.entityKind) \land
((td = td') \lor \text{isSubtype}_0(td, td'))))\)

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**

\[
\begin{align*}
& \langle \text{composite state type heading} \rangle(v, q, n, cPar, vc, \langle \text{specialization} \rangle(\langle \text{type expression} \rangle(base, actPar), *), p) \\
& \text{provided getUnboundFormalContextParameterList}_0(\text{actParams}, \text{base}.) \nonumber \\
& \quad \# \text{ empty} \\
& \quad =11=\Rightarrow \\
& \quad \langle \text{composite state type heading} \rangle(v, q, n, nCPar, vc, \langle \text{specialization} \rangle(\langle \text{type expression} \rangle(b, nActPar), \text{undefined}), p) \\
& \quad \text{endlet})
\end{align*}
\]

\[
\begin{align*}
& \langle \text{agent type additional heading} \rangle(cPar, vc, \\
& \quad \langle \text{agent additional heading} \rangle(\langle \text{specialization} \rangle(\langle \text{type expression} \rangle(base, actPar), *), p)) \\
& \text{provided getUnboundFormalContextParameterList}_0(\text{actParams}, \text{base}.) \nonumber \\
& \quad \# \text{ empty} \\
& \quad =11=\Rightarrow \\
& \quad \langle \text{agent type additional heading} \rangle(nCPar, vc, \\
& \quad \langle \text{agent additional heading} \rangle(\langle \text{specialization} \rangle(\langle \text{type expression} \rangle(base, nActPar), \text{undefined}), p)) \\
& \quad \text{endlet})
\end{align*}
\]

\[
\begin{align*}
& \langle \text{procedure heading} \rangle(v, q, n, cPar, vc, \langle \text{specialization} \rangle(\langle \text{type expression} \rangle(base, actPar), *), p, r, x) \\
& \text{provided getUnboundFormalContextParameterList}_0(\text{actParams}, \text{base}.) \nonumber \\
& \quad \# \text{ empty} \\
& \quad =11=\Rightarrow \\
& \quad \langle \text{procedure heading} \rangle(v, q, n, nCPar, vc, \\
& \quad \langle \text{specialization} \rangle(\langle \text{type expression} \rangle(b, nActPar), \text{undefined}), p, r, x) \\
& \quad \text{endlet})
\end{align*}
\]

\[
\begin{align*}
& \langle \text{signal definition item} \rangle(n, cPar, vc, \langle \text{specialization} \rangle(\langle \text{type expression} \rangle(base, actPar), *), p) \\
& \text{provided getUnboundFormalContextParameterList}_0(\text{actParams}, \text{base}.) \nonumber \\
& \quad \# \text{ empty}
\end{align*}
\]
(let nCPar = cPar \cap
getUnboundFormalContextParameterList(actParams, base.refersto, s\langle formal context parameter\rangle)
in
let nActPar =
completeFormalContextParameter(actParams, base.refersto, s\langle formal context parameter\rangle)
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(fcp: FORMALCONTEXTPARAMETERS): ENTITYDEFINITION = _set
case fcp of
| <agent type context parameter> =>
  if (fcp.s\langle agent type constraint\rangle\in <identifier>) then
    getEntityDefinition(fcp.s\langle agent type constraint\rangle, <identifier>, agent type)
  else undefined
  endif
| <agent context parameter> =>
  if (fcp.s\langle agent constraint\rangle\in <identifier>) then
    getEntityDefinition(fcp.s\langle agent constraint\rangle, <identifier>, agent type)
  else undefined
  endif
| <procedure context parameter> =>
  if (fcp.s\langle procedure constraint\rangle\in <identifier>) then
    getEntityDefinition(fcp.s\langle procedure constraint\rangle, procedure)
  else undefined
  endif
| <composite state type context parameter> =>
  if (fcp.s\langle composite state type constraint\rangle\in <identifier>) then
    getEntityDefinition(fcp.s\langle composite state type constraint\rangle, state type)
  else undefined
  endif
| <signal context parameter gen name> =>
  if (fcp.s\langle signal constraint\rangle\in <identifier>) then
    getEntityDefinition(fcp.s\langle signal constraint\rangle, signal)
  else undefined
  endif
| <sort context parameter> =>
  if (fcp.s\langle sort constraint\rangle\in <sort>) then
    getEntityDefinition(fcp.s\langle sort constraint\rangle, type)
  else undefined
  endif
| <interface context parameter gen name> =>
  if (fcp.s\langle interface constraint\rangle \neq undefined) then
    getEntityDefinition(fcp.s\langle interface constraint\rangle, interface)
  else undefined
  endif
otherwise undefined
endcase
2.5.7.1 Agent Type Context Parameter

Concrete Syntax

\[
\text{<agent type context parameter>} ::
\]
\[
\text{<agent kind>} \text{ <agent type <name>} \text{ [<agent type constraint>]}
\]

\[\text{<agent kind>} = \text{process} | \text{block}\]

\[\text{<agent type constraint>} = \text{<agent type <identifier>} | \text{<agent signature>}\]

Conditions on Concrete Syntax

\[
\forall fcp \in \text{<agent type context parameter>}: \forall acp \in \text{<actual context parameter>}: \text{isContextParameterCorresponded}(acp, fcp) \Leftrightarrow \begin{align*}
\text{(let } t0 = \text{getEntityDefinition}(acp, \text{agent type}) \text{ in } \\
\text{(3}td, element(\text{<agent type definition}>): (td = fcp, \text{contextParameterAtleastDefinition}0) \land \\
\text{(td}_1 = \text{agentLocalFormalParameterList}_0 = \text{empty}) \land \text{isSubtype}(td,td') \lor \\
\text{(td}_1 = fcp, \text{entityKind}_0) \land \\
\text{(let } pl = td,\text{agentFormalParameterList}0 \text{ in } \\
\text{let sl = fcp,\text{agent type constraint}.agentSignatureSortList}0 \text{ in } \\
\text{(pl}_1 = sl,\text{length}) \land \\
\text{(\forall i \in 1..pl}_1,\text{length: isSameSort}(pl,\text{kind}_0, \text{sl}_1,\text{kind}_0))\text{ endlet) endlet)}
\end{align*}
\]

An actual agent type parameter must be a subtype of the constraint agent type (\text{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 \text{<sort>s} of the \text{<agent signature>}.

Auxiliary Functions

Get the sort list defined in an \text{<agent signature>}.

\[
\text{agentSignatureSortList}(\text{as}: \text{<agent signature>}:) : \text{<sort>*} = \text{def} (\text{as}_1,\text{<sort>},\text{-seq})
\]

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

\[
\text{agentFormalParameterList}_0(t0 : \text{<agent type definition}> : \text{<agent definition>}) : \text{<name>*} = \text{def} (t0,\text{agentLocalFormalParameterList}0 \cap t0,\text{agentInheritedFormalParameterList}0)
\]

\[
\text{agentLocalFormalParameterList}_0(t0 : \text{<agent type definition>}) : \text{<name>*} = \text{def} (t0,\text{agentLocalFormalParameterList}0 \cap t0,\text{agentInheritedFormalParameterList}0)
\]

\[
\text{agentInheritedFormalParameterList}_0(t0 : \text{<agent type definition>}) : \text{<name>*} = \text{def} (t0,\text{specialization}_0 \cap t0,\text{agentInheritedFormalParameterList}0)
\]

\[
\text{agentFormalParameterList}_0(t0 : \text{<agent type definition>}) : \text{<name>*} = \text{def} (t0,\text{specialization}_0 \cap t0,\text{agentFormalParameterList}0)
\]

Determine if a formal context parameter corresponds to an actual context parameter.
isContextParameterCorresponded(fcp: FORMALCONTEXTPARAMETER0, acp: <actual context parameter>): BOOLEAN
let fcpl = fcp.surroundingScopeUnit0.formalContextParameterList0 in
let acpl = parentAS0ofKind(acp, <type expression>).actualContextParameterList0 in
(fcp.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(fcp, acp) ⇒
(let td = getEntityDefinition(acp, agent) in
  let td' = fcp.contextParameterAtleastDefinition0 in
  if(fcp.s.<agent constraint>.s-atleast ≠ undefined) ⇒
    isSubtype(td, td') ∧ (td.agentLocalFormalParameterList0 = empty)) ∧
  if((fcp.s.<agent constraint> s-atleast = undefined) ⇒ (td = td')) ∧
  if((fcp.s.<agent constraint> ∈ <agent signature>) ⇒
    (getEntityDefinition(acp, agent).entityKind0 = fcp.entityKind0) ∧
    (let pl = td.agentFormalParameterList0 in
     let sl = fcp.s.<agent constraint>.agentSignatureSortList0 in
     if(pl.length = sl.length) ∧
     (∀i∈ 1..pl.length: isSameSort(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(fcp, acp) ⇒
(let td = getEntityDefinition(acp, procedure) in
  if((fcp.s.<procedure constraint> ∈ <identifier>) ⇒
    (let td' = fcp.contextParameterAtleastDefinition0 in
     isDirectSubType(td, td') 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(fcp, acp) ⇒

(let ps = getEntityDefinition(acp, remote procedure).s<procedure signature> in
let ps' = fcp.s<procedure signature> in

isSameProcedureSignature(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(ps, ps': <procedure signature>): BOOLEAN =def

let fpl = ps.procedureSignatureParameterList in
let fpl' = ps'.procedureSignatureParameterList in

(fpl.length = fpl'.length) ∧

(∀i ∈ 1..fpl.length:

(isSameSort(fpl[i].s<sort>, fpl'[i].s<sort>) ∧

(isSameParameterKind(fpl[i].s<parameter kind>, fpl'[i].s<parameter kind>)) ∧

(isSameResult(ps.s<result>, ps'.s<result>) ∧

(isSameRaises(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(fcp, acp) ⇒
(let sd = getEntityDefinition(acp, signal) in
((fcp.s-<signal constraint> ∈ <identifier>) ⇒
(let sd' = fcp.contextParameterAtleastDefinition0 in
isSubtype0(sd, sd') endlet)) ∧
((fcp.s-<signal constraint> ∈ <signal signature>) ⇒
isSameSortList0(sd.s-<sort>-seq, fcp.s-<signal constraint>.s-<sort>-seq )) endlet)

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

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(fcp, acp) ⇒
(let vd = getEntityDefinition(acp, variable) in
isSameSort0(fcp.parentAS0.s-<sort>, vd.s-<sort>) endlet)

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

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(fcp, acp) ⇒
An actual parameter must identify a <remote variable definition> of the same sort.

2.5.7.8 Timer Context Parameter

Concrete Syntax

\[
\text{<timer context parameter> :: <timer context parameter gen name> +} \\
\text{<timer context parameter gen name> :: <timer name> [<timer constraint>]}
\]

Conditions on Concrete Syntax

\[
\forall fcp \in \text{<timer context parameter gen name> : } \forall acp \in \text{<actual context parameter>}: \\
\text{isContextParameterCorresponded}(fcp, acp) \Rightarrow \\
(\text{let } \text{td} = \text{getEntityDefinition}_{acp, \text{timer}} \text{ in} \\
\text{isSameSortList}_{fcp.s.<sort>., \text{td}s.<sort>.-seq, \text{td}s.<sort>.-seq})
\]

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

\[
\text{<synonym context parameter> :: <synonym context parameter gen name> +} \\
\text{<synonym context parameter gen name> :: <synonym name> <synonym constraint>}
\]

Conditions on Concrete Syntax

\[
\forall fcp \in \text{<synonym context parameter gen name> : } \forall acp \in \text{<actual context parameter>}: \\
\text{isContextParameterCorresponded}(fcp, acp) \Rightarrow \\
(\text{let } \text{sd} = \text{getEntityDefinition}_{acp, \text{synonym}} \text{ in} \\
\text{isSameSort}_{\text{sd}s.<sort>, \text{fcp}s.<synonym constraint>.s.<sort>})
\]

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

\[
\text{<sort context parameter> :: [ <data type kind> ] <sort name> [ <sort constraint> ]}
\]

Conditions on Concrete Syntax

\[
\forall fcp \in \text{<sort context parameter> : } \forall acp \in \text{<actual context parameter>}: \\
\text{isContextParameterCorresponded}(fcp, acp) \Rightarrow \\
(\text{let } \text{sd} = \text{getEntityDefinition}_{acp, \text{synonym}} \text{ in} \\
\text{isSameSort}_{\text{sd}s.<sort>, \text{fcp}s.<synonym constraint>.s.<sort>})
\]

An actual synonym must be a constant expression of the same sort as the sort of the constraint.
Conditions on Concrete Syntax

∀fcp <sort context parameter> : ∀acp <actual context parameter>:
  isContextParameterCorresponded(fcp, acp) ∧ (fcp.s <sort constraint> ≠ undefined) ⇒
  (let td = getEntityDefinition(acp, type).derivedDataType in
    (fcp.s <sort constraint> ∈ <sort> ⇒
    (let td' = fcp.contextParameterAtleastDefinition in
      isSubtype(td, td') ∧
      (∀td1 ∈ <data type definition> ∪ {td}; isSubtype(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'.surroundingScopeUnit0 = td) ∧ isSameLiteralSignature(ls, ls') ∧
  (∀os ∈ <operation signature> : (os.parentAS0 = fcp.s <sort constraint>) ⇒
  ∃os' ∈ <operation signature> :
  (os'.surroundingScopeUnit0 = td) ∧ (os.visibility0 = os'.visibility0) ∧
  (os.visibility0 = os'.visibility0) ∧ isSameOperationSignature(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 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.

\[
\text{derivedDataType}(sd: <syntype definition> ⊎ <data type definition>): <data type definition> = \text{def}
\]

if (sde <syntype definition>) then sd.parentDataType
else sd
endif

Get the parent data type definition of a syntype definition.

\[
\text{parentDataType}(sd: <syntype definition>): <data type definition> = \text{def}
\]

if (sd.s <parent sort identifier> = undefined) then sd
else
  let pd = getEntityDefinition(sd.s <parent sort identifier>, type) in
  if (pde <data type definition>) then pd
  else pd.parentDataType
endif
endif

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

\[
\text{isSameLiteralSignature}(ls: <literal signature>, ls': <literal signature>): \text{BOOLEAN} = \text{def}
\]

((ls ∈ <literal name>) ∧ (ls' ∈ <literal name>) ⇒ (ls = ls')) ∧
((ls ∈ <name class literal>) ∧ (ls' ∈ <name class literal>) ⇒ isSameNameClassLiteral(ls, ls') ∧
((ls ∈ <named number>) ∧ (ls' ∈ <named number>) ⇒
  (ls.s <literal name> = ls'.s <literal name>) ∧
  (ls.s <simple expression>.value0 = ls'.s <simple expression>.value0))
Determine if two <operation signature> are the same.

\[\text{isSameOperationSignature}(os: <\text{operation signature}>, os': <\text{operation signature}>): \text{BOOLEAN} = \text{def} (os.s-implicit \in \text{operation name} \Rightarrow os.s-implicit = os'.s-implicit) \land (os.s-implicit \in \text{name class operation} \Rightarrow \text{isSameNameClassOperation}(os.s-implicit, os'.s-implicit)) \land (\text{let} fpl = os.\text{operationSignatureParameterList}_0 \text{ in} \text{let} fpl' = os'.\text{operationSignatureParameterList}_0 \text{ in} (fpl.length = fpl'.length) \land (\forall i \in 1..fpl.length:\ (fpl[i].s-<\text{formal parameter}>=fpl'[i].s-<\text{formal parameter}>) \land \text{isSameSort}(fpl[i].s-<\text{sort}>, fpl'[i].s-<\text{sort}>) \land \text{isSameResult}(os.s-<\text{result}>, os'.s-<\text{result}>) \land \text{isSameRaises}(os.s-<\text{raises}>, os'.s-<\text{raises}>) \land \text{isacontextparameter}(\text{exception context parameter})

2.5.7.11 Exception Context Parameter

Concrete Syntax

\[<\text{exception context parameter}>::=\]
\[<\text{exception context parameter gen name}>^+\]

\[<\text{exception context parameter gen name}> ::=<\text{exception name}>[<\text{exception constraint}>]\]

\[<\text{exception constraint}> = <\text{sort}>^+\]

Conditions on Concrete Syntax

\[\forall fcp \in <\text{exception context parameter gen name}> : \forall acp \in <\text{actual context parameter}>:\]
\[\text{isContextParameterCorresponded}(fcp, acp) \land (fcp.s-<\text{exception constraint}> \neq \text{undefined}) \Rightarrow (\text{let} ed = \text{getEntityDefinition}(acp, \text{exception}) \text{ in} \text{isSameSortList}(fcp.s-<\text{exception constraint}>.s-<\text{sort}>, ed.s-<\text{sort}>.seq) \text{ endlet})\]

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

2.5.7.12 Composite State Context Parameter

Concrete Syntax

\[<\text{composite state type context parameter}>::=\]
\[<\text{composite state type name}>[<\text{composite state type constraint}>]\]

\[<\text{composite state type constraint}> =<\text{identifier}> | <\text{composite state type signature}>\]

\[<\text{composite state type signature}> = <\text{sort}>^+\]

Conditions on Concrete Syntax

\[\forall fcp \in <\text{composite state type context parameter}> : \forall acp \in <\text{actual context parameter}>:\]
\[\text{isContextParameterCorresponded}(fcp, acp) \land (fcp.s-<\text{composite state type constraint}> \neq \text{undefined}) \Rightarrow (\text{let} td = \text{getEntityDefinition}(acp, \text{state type}) \text{ in} \text{isSameSortList}(fcp.s-<\text{composite state type constraint}>.s-<\text{sort}>,.seq, td.s-<\text{sort}>.seq) \text{ endlet})\]

\[\text{(td.agentLocalFormalParameterList}_0 = \text{empty}) \land \text{isSubtype}(td, td') \text{ endlet})\]
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 (\texttt{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 \texttt{<sort>}s of the \texttt{<composite state type signature>}.

### 2.5.7.13 Gate Context Parameter

**Concrete Syntax**

\[
\langle \text{gate context parameter} \rangle :: \langle \text{gate} \rangle \langle \text{gate constraint} \rangle \begin{array}{c}
\text{[\text{<gate constraint>}]}
\end{array}
\]

**Conditions on Concrete Syntax**

\[
\forall fcp \in \langle \text{gate context parameter} \rangle : \forall acp \in \langle \text{actual context parameter} \rangle : \\
\text{isContextParameterCorresponded}(fcp, acp) \Rightarrow \\
\begin{array}{c}
\text{(let gd = getEntityDefinition(acp, gate) in)} \\
(gd.s-<gate> = fcp.s-<gate>) \land \\
(\forall gc \in \langle \text{gate constraint} \rangle : \forall gc' \in \langle \text{gate constraint} \rangle : \\
(gc.parentAS0 = gd) \land (gc'.parentAS0 = fcp) \land \\
(gc.direction = \text{out}) \land (gc'.direction = \text{out}) \Rightarrow \\
gc'.s-<sort>-\text{seq} . \text{signalSets} = gc.s-<sort>-\text{seq} . \text{signalSets} \land \\
(gc.direction = \text{in}) \land (gc'.direction = \text{in}) \Rightarrow \\
gc.s-<sort>-\text{seq} . \text{signalSets} = gc'.s-<sort>-\text{seq} . \text{signalSets}
\end{array}
\]

An actual gate parameter must identify a gate definition. Its outward gate constraint must contain all elements mentioned in the \texttt{<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 \texttt{<signal list>} of the actual gate parameter.

### 2.5.7.14 Interface Context Parameter

**Concrete Syntax**

\[
\langle \text{interface context parameter} \rangle = \langle \text{interface context parameter gen name} \rangle +
\]

\[
\langle \text{interface context parameter gen name} \rangle :: \langle \text{interface} \rangle \langle \text{interface constraint} \rangle \begin{array}{c}
\text{[\text{<interface constraint>}]}
\end{array}
\]

**Conditions on Concrete Syntax**

\[
\forall fcp \in \langle \text{interface context parameter gen name} \rangle : \forall acp \in \langle \text{actual context parameter} \rangle : \\
\text{isContextParameterCorresponded}(fcp, acp) \Rightarrow \\
(\exists td \in \langle \text{interface definition} \rangle : \\
(td = \text{getEntityDefinition(acp.s-<identifier>, interface)}) \land \\
\text{isSubtype}(td, fcp.contextParameterAtleastDefinitions))
\]

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 (\texttt{atleast <interface identifier>}).
2.5.8 Specialization

2.5.8.1 Adding properties

Concrete Syntax

\[ <\text{specialization}> ::= <\text{type expression}> \]

Mapping to Abstract Syntax

\[ | <\text{specialization}> (x) => Mapping(x) | \]

Auxiliary Functions

The function \( \text{specialization}_0 \) is used to get the \( <\text{specialization}> \) part of an entity definition.

\[
\text{specialization}_0(\text{def} \cdot \text{ENTITYDEFINITION}_0): <\text{specialization}> =_{\text{def}} \text{take}( \{ \text{se} <\text{specialization}> \cup <\text{data type specialization}> \cup <\text{interface specialization}>: \text{s.surroundingScopeUnit}_0 = \text{def} \})
\]

2.5.8.2 Virtual Type

Concrete Syntax

\[ <\text{virtuality}> = \text{virtual} | \text{redefined} | \text{finalized} \]

\[ <\text{virtuality constraint}> ::= <\text{identifier}> \]

Conditions on Concrete Syntax

\[ \forall t\in <\text{block type definition}> \cup <\text{process type definition}> \cup <\text{procedure definitions}> \cup <\text{composite state type definition}>:
\]

\[ \text{isVirtualType}_0(t) \Rightarrow t.virtualTypeAtleastDefinition_0.\text{entityKind}_0 = t.entityKind_0 \]

Every virtual type has associated a virtuality constraint which is an \( <\text{identifier}> \) of the same entity kind as the virtual type.

\[ \forall t\in <\text{block type definition}> \cup <\text{process type definition}> \cup <\text{procedure definitions}> \cup <\text{composite state type definition}>:
\]

\[ t.d.isVirtualType_0 \Rightarrow
\]

\[ \neg(t.d.isParameterizedType_0) \wedge \neg(isParameterizedType_0(t.d.virtualTypeAtleastDefinition_0)) \]

A virtual type and its constraints cannot have context parameters.

\[ \forall v\in <\text{virtuality constraint}>: \text{isVirtualType}_0(v.c.surroundingScopeUnit}_0) \]

Only virtual types may have \( <\text{virtuality constraint}> \) specified.

\[ \forall r\in <\text{block type reference}> \cup <\text{process type reference}> \cup <\text{composite state type reference}> \cup <\text{procedure reference}>:
\]

\[ \forall d\in <\text{block type definition}> \cup <\text{process type definition}> \cup <\text{composite state type definition}> \cup <\text{procedure definitions}>:
\]

\[ (r.referencedDefinition_0 = d) \wedge (r.virtuality_0 \neq \text{undefined}) \wedge (d.virtuality_0 \neq \text{undefined}) \Rightarrow
\]

\[ (r.virtuality_0 = d.virtuality_0) \]

If \( <\text{virtuality}> \) is present in both the reference and the referenced definition, then they must be equal. If \( <\text{procedure preamble}> \) is present in both procedure reference and in the referenced procedure definition, they must be equal.

\[ \forall t\in <\text{block type definition}> \cup <\text{process type definition}>: \text{isVirtualType}_0(t) \Rightarrow
\]

\[ (\text{let } t' = t.d.virtualTypeAtleastDefinition_0 \text{ in}
\]

\[ (\text{let } fpl = t.d.agentFormalParameterList_0 \text{ in}
\]

\[ \text{let } fpl' = t.d.agentFormalParameterList_0 \text{ in}
\]

\[ \text{isSameAgentFormalParameterList}(fpl, fpl') \text{ endlet}) \wedge \]

\[ \]
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.

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

Auxiliary Functions

Mapping to Abstract Syntax

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

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

Mapping involving inherits must be a subtype of the constraint.

Determining Virtuality

In the case of an implicit constraint, redefinition involving inherits must be a subtype of the constraint.
isRedefinedType_{th}(td: <block type definition>)\cup<process type definition>\cup<procedure definitions>\cup<composite state type definition>): BOOLEAN =def

td.virtuality\in\{\text{redefined, finalized}\}

Get the virtuality for a definition.

\texttt{virtuality}(td: DefinitionAS_{th0}): \{\text{virtual, redefined, finalized}\} =def
\begin{aligned}
\text{case } d \text{ of } \\
| <\text{block type definition}> & \Rightarrow \ \\
| d.s.<\text{block type heading}>.s.<\text{type preamble}>.s.<\text{virtuality}> & \Rightarrow \\
| <\text{process type definition}> & \Rightarrow \\
| d.s.<\text{process type heading}>.s.<\text{type preamble}>.s.<\text{virtuality}> & \Rightarrow \\
| <\text{composite state type definition}> & \Rightarrow \\
| d.s.<\text{composite state type heading}>.s.<\text{virtuality}> & \Rightarrow \\
| <\text{procedure definition}> & \Rightarrow \\
| d.s.<\text{procedure heading}>.s.<\text{procedure preamble}>.s.<\text{type preamble}>.s.<\text{virtuality}> & \Rightarrow \\
| <\text{operation definition}> & \Rightarrow \\
| d.s.<\text{operation heading}>.s.<\text{operation preamble}>.s.<\text{virtuality}> & \Rightarrow \\
| <\text{interface definition}> & \Rightarrow \\
| d.s.<\text{interface reference}>.s.<\text{virtuality}> & \Rightarrow \\
| <\text{interface reference}> & \Rightarrow d.s.<\text{virtuality}> \\
| <\text{operation signature}> & \Rightarrow d.s.<\text{operation preamble}>.s.<\text{virtuality}> \\
| <\text{start}>.\cup<\text{input part}>.\cup<\text{priority input}>.\cup<\text{save part}>.\cup<\text{spontaneous transition}>.\cup<\text{continuous signal}>.\cup<\text{connect part}>.\cup<\text{handle}>.\cup<\text{default initialization}> & \Rightarrow d.s.<\text{virtuality}> \\
| <\text{statement list}> & \Rightarrow d.parentAS_{th0}.s.<\text{virtuality}> \\
\text{otherwise } undefined & \Rightarrow \\
\text{endcase}
\end{aligned}

Get the entity definition referred by a \texttt{<virtuality constraint>},

virtualTypeAtleastDefinition_{th}(td: <block type definition>)\cup<process type definition>\cup<procedure definitions>\cup<composite state type definition>): <block type definition> =def

\textbf{let} \texttt{vc}=\texttt{take}(\{vc:\texttt{<virtuality constraint>}: \texttt{vc.surroundingScopeUnit}_{th}=td\}) \textbf{in}
\begin{aligned}
\text{if } vc \neq undefined & \texttt{then } \texttt{getEntityDefinition}(vc, td.entityKind_{th}) \texttt{else } td & \texttt{endif} \\
\texttt{endlet}
\end{aligned}

Determine if two agent formal parameter lists are the same.

\texttt{isSameAgentFormalParameterList}_{th}(fpl: <name>*, fpl':<name>*): BOOLEAN =def
\begin{aligned}
(fpl.length = fpl'.length) & \land \\
(\forall i \in 1..fpl.length: (fpl[i] = fpl'[i]) \land \texttt{isSameSort}_{th}(fpl[i].parentAS_{th0}.s.<\text{sort}>, fpl'[i].parentAS_{th0}.s.<\text{sort}>))
\end{aligned}

Determine if two \texttt{<gate> in definition}s are the same.

\texttt{isSameGate}_{th}(gd: <gate in definition>, gd': <gate in definition>): BOOLEAN =def
\begin{aligned}
\text{if } (gd.e.<\text{textual gate definition}>\land gd'e.<\text{textual gate definition}> & \texttt{then} \\
(gd.s.<\text{gate}> = gd'.s.<\text{gate}>) & \land \\
(\forall gc\in gd.s.<\text{gate constraint}>: \exists gc'\in gd'.s.<\text{gate constraint}>:\ \\
(gc.s.<\text{direction}> = gc'.s.<\text{direction}>) & \land \\
(gc.s.<\text{textual endpoint constraint}> = gc'.s.<\text{textual endpoint constraint}>) & \land \\
\texttt{isSameSortList}_{th}(gc.s.<\text{sort}>.seq, gc'.s.<\text{sort}>.seq)) & \land \\
(\forall gc\in gd'.s.<\text{gate constraint}>: \exists gc\in gd.s.<\text{gate constraint}>:\ \\
(gc.s.<\text{direction}> = gc'.s.<\text{direction}>) & \land \\
(gc.s.<\text{textual endpoint constraint}> = gc'.s.<\text{textual endpoint constraint}>) & \land \\
\texttt{isSameSortList}_{th}(gc.s.<\text{sort}>.seq, gc'.s.<\text{sort}>.seq)) & \land \\
\texttt{else if } (gd.e.<\text{textual interface gate definition}>\land gd'e.<\text{textual interface gate definition}> & \texttt{then} \\
gd.s.<\text{identifier}>=gd'.s.<\text{identifier}> & \texttt{else } False & \texttt{endif}
\end{aligned}
Determine if two <interface definition>s are the same.

\[
\text{isSameInterface}(id; \text{<interface definition>}, id'; \text{<interface definition>}): \text{BOOLEAN} = \text{def}
\]

\[
(id.\text{virtuality}_0 = id'.\text{virtuality}_0) \land
(id.\text{entityName}_0 = id'.\text{entityName}_0) \land
(id.\text{entityDefinitionSet}_0 = id'.\text{entityDefinitionSet}_0)
\]

Get all the entity definitions defined in a scope unit.

\[
\text{entityDefinitionSet}(su; \text{SCOPEUNIT}_0): \text{ENTITYDEFINITION}_0 = \text{def}
\]

\[
\{ ed \in \text{ENTITYDEFINITION}_0: \text{isDefinedIn}(ed, su) \}
\]

Get all the interface definitions defined in a scope unit.

\[
\text{interfaceDefinitionSet}(d; \text{SCOPEUNIT}_0): \text{<interface definition>-set} = \text{def}
\]

\[
\{ fd \in \text{<interface definition>}: \text{isDefinedIn}(fd, d) \}
\]

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

\[
\text{stateConnectionPointSet}(td; \text{<composite state type definition>} \cup \text{<composite state> }): \text{<state connection points>-set} = \text{def}
\]

\[
\{ scp \in \text{<state connection points>}: (scp.\text{parentAS}_0 = td) \land
(\exists td' \in \text{<composite state type definition>}: \text{isSubtype}(td, td') \land (scp.\text{parentAS}_0 = td')) \}
\]

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

\[
\text{isSameStateConnectionPoint}(scp; \text{<state connection points>}, scp': \text{<state connection points>}): \text{BOOLEAN} = \text{def}
\]

\[
\{ n \in \text{<name>}: \text{isAncestorAS}(scp, n) \} = \{ n' \in \text{<name>}: \text{isAncestorAS}(scp', n) \}
\]

Determine if two procedure formal parameter lists are the same.

\[
\text{isSameProcedureFormalParameterList}(fpl; \text{<name>*}, fpl': \text{<name>*}): \text{BOOLEAN} = \text{def}
\]

\[
(fpl.\text{length} = fpl'.\text{length}) \land
(\forall i \in 1..fpl.\text{length}: (fpl[i].\text{parentAS}_0.\text{parentAS}_0.\text{s}-<\text{parameter kind}> = fpl'[i].\text{parentAS}_0.\text{parentAS}_0.\text{s}-<\text{parameter kind}>) \land
(fpl[i] = fpl'[i]) \land \text{isSameSort}(fpl[i].\text{parentAS}_0.\text{s}-<\text{sort}>, fpl'[i].\text{parentAS}_0.\text{s}-<\text{sort}>)
\]

Get the entity definition specialised by a virtual type.

\[
\text{virtualTypeInheritsDefinition}(td; \text{<block type definition>} \cup \text{<process type definition>} \cup \text{<procedure definitions>} \cup \text{<composite state type definition>}: \text{<block type definition}> \cup \text{<process type definition>} \cup \text{<procedure definitions}> \cup \text{<composite state type definition>} = \text{def}
\]

\[
\text{let } sp = \text{td.specialization}_0 \text{ in}
\]

\[
\text{if } (sp \neq \text{undefined}) \text{ then } sp.\text{s}-<\text{type expression}>.\text{baseType}_0 \text{ else}
\]

\[
\text{let } wc = \text{take}(\{ \text{vc} \in \text{virtuality constraint}: \text{vc.\text{surroundingScopeUnit}_0 = id'} \}) \text{ in}
\]

\[
\text{case } \text{td.\text{virtuality}_0} \text{ of}
\]

\[
| \text{virtual} => \text{if } (wc = \text{undefined}) \text{ then } \text{undefined} \text{ else } \text{td.\text{virtualTypeAtleastDefinition}_0} \text{ endif}
\]

\[
| \text{redefined} =>
\]

\[
\text{if } (wc = \text{undefined}) \text{ then } \text{td.\text{superCounterpart}_0.\text{virtualTypeAtleastDefinition}_0} \text{ else } \text{td.\text{virtualTypeAtleastDefinition}_0} \text{ endif}
\]

\[
\text{endcase}
\]

\[
\text{endif}
\]

\[
\text{endlet}
\]

For a given entity definition, get the counterpart in the super type of the surrounding scope unit.
superCounterpart(\(td: \text{<block type definition>} \cup \text{<procedure definitions}> \cup \text{<operation definition>} \cup \text{<procedure definitions}> \cup \text{<operation signature}>\)):

\((\text{<block type definition>} \cup \text{<procedure definitions>} \cup \text{<operation definition>} \cup \text{<procedure definitions>} \cup \text{<operation signature>}\) \\\text{def}

\{\{td'\in <\text{block type definition}> <\text{procedure definitions}> <\text{operation definition}> <\text{procedure definitions}> <\text{operation signature}> \}:

\( isSuperCounterpart(\text{td', td}) \)

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

\( isSuperCounterpart(\text{td}: \text{DefinitionAS0}, \text{td'}: \text{<block type definition>} \cup \text{<procedure definitions>} \cup \text{<composite state type definition>} \cup \text{<operation definition>} \cup \text{<procedure definitions>})\):

\( BOOLEAN =\text{def} (\text{td.name}_0 = \text{td'}.name_0) \land (\text{td.kind}_0 = \text{td'}.kind_0) \land (\text{td.virtuality}_0 \in \{\text{virtual, redefined}\}) \land (\text{td'}.virtuality_0 \in \{\text{redefined, finalized}\}) \land \text{isDirectSubType}(\text{td'}.\text{surroundingScopeUnit}_0, \text{td}.\text{surroundingScopeUnit}_0) \)

### 2.5.8.3 Virtual Transitions/Save

#### Conditions on Concrete Syntax

\[ \forall \text{ad} \in \text{<agent definition>} \cup \text{<textual typebased agent definition>} \cup \text{<composite state>} \cup \text{<typebased composite state>}: \]

\[ (\forall \text{se} \in \text{<start>}: s \text{ad}.\text{startSet}_0 = \text{s.virtuality}_0 = \text{undefined}) \land \]

\[ (\forall \text{se} \in \text{<state>}: (s \text{ad}.\text{stateSet}_0 = (s.\text{ad}.\text{startSet}_0 = (s.\text{s}.<\text{input part}>.\text{virtuality}_0 = \text{undefined}) \land \]

\[ (s.\text{s}.<\text{priority input}>.\text{virtuality}_0 = \text{undefined}) \land \]

\[ (s.\text{s}.<\text{save part}>.\text{virtuality}_0 = \text{undefined}) \land \]

\[ (s.\text{s}.<\text{spontaneous transition}>.\text{virtuality}_0 = \text{undefined}) \land \]

\[ (s.\text{s}.<\text{continuous signal}>.\text{virtuality}_0 = \text{undefined}) \]

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

\[ \forall \text{se} \in \text{<state>}: (|\{\text{se} \in \text{spontaneous transition}>: (\text{s.parentAS0} = \text{s}) \land (\text{s.virtuality}_0 = \text{undefined})|) \leq 1 \]

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

\[ (\forall \text{ip} \in \text{<input part>} : \]

\[ (\text{ip.virtuality}_0 = \text{undefined}) \Rightarrow \text{ip.s}<\text{in list}_0 \in \text{asterisk input list}) \land \]

\[ (\forall \text{sp} \in \text{<save part>} : \]

\[ (\text{sp.virtuality}_0 \neq \text{undefined}) \Rightarrow \text{sp.s}<\text{in list}_0 \in \text{asterisk save list}) \]

An input or save with \text{<virtuality>} must not contain \text{<asterisk>}.

#### Auxiliary Functions

Get the set of \text{<start>} defined in a given definition.

\[ \text{startSet}(\text{td}: \text{<agent definition>} \cup \text{<textual typebased agent definition>} \cup \text{<agent type definition>} \cup \text{<composite state>} \cup \text{<typebased composite state>} \cup \text{<composite state type definition>} \cup \text{<textual typebased state partition definition>} \cup \text{<state partition>} \cup \text{<procedure definitions>})\): \text{<start> set} \text{def}

\[ \text{case td of} \]

\[ | \text{<agent definition>} \cup \text{<agent type definition>} \cup \text{<composite state type definition>} \cup \text{<procedure definitions>} \Rightarrow \text{td}.\text{localStartSet}_0 \cup \text{td.inheritedStartSet}_0 \]

\[ | \text{<textual typebased agent definition>} \cup \text{<textual typebased state partition definition>} =\]

\[ \text{let te}=\text{take} \{\{\text{te} <\text{type expression}> : \text{te.parentAS0}.\text{parentAS0} = \text{td}\}\} \text{ in} \]

\[ \text{te}.\text{baseType}_0.\text{startSet}_0 \]

\[ \text{endlet} \]

\[ | \text{<composite state> } \Rightarrow \]

\[ \text{if td.s}<\text{<composite state structure>} <\text{explicit} \in <\text{composite state body}> \text{ then} \]

\[ \{\text{s} <\text{start}> : \text{s.parentAS0} = \text{td.s}<\text{<composite state structure>} <\text{explicit}\} \]
Get the set of \textit{<state>} defined in a given definition.

\textbf{localStartSet} \textbf{(id: <agent definition> \cup <agent type definition> \cup <composite state type definition> \cup <procedure definitions>): <start> = def}

\textbf{case id of}

| \textbf{<agent definition> \cup <agent type definition> \Rightarrow}
| if \textbf{id.s} <agent structure> \textbf{.s-implicit} \in <agent body> then
| \{ \textbf{id.s} <agent structure> \textbf{.s-implicit}.s <-state partition>.startSet_{0} \}
| else \textbf{id.s} <agent structure> \textbf{.s-implicit}.s <-state partition>.startSet_{0} \end{\textbf{case}}

\textbf{inheritedStartSet} \textbf{(id: <agent definition> \cup <agent type definition> \cup <composite state type definition> \cup <procedure definitions>): <start> = def}

\textbf{let sp = id.specialization_{0} in}

| if \textbf{sp} = undefined then \emptyset
| else \textbf{sp} <type expression>.baseType_{0}.startSet_{0} \end{\textbf{let}}

\textbf{endlet}

\textbf{stateSet} \textbf{(id: <agent definition> \cup <textual typebased agent definition> \cup <agent type definition> \cup <composite state> \cup <typebased composite state> \cup <composite state type definition> \cup <procedure definitions>): <state> = def}

\textbf{case id of}

| \textbf{<agent definition> \cup <agent type definition> \cup <composite state type definition> \cup <procedure definitions>} =>
| \textbf{id.localStartSet_{0}.id.inheritedStartSet_{0}}
| \textbf{| <textual typebased agent definition> \cup <textual typebased state partition definition> =>}
| let \textbf{te = take(\{ te \in <type expression>: te.parentAS0.parentAS0 = id \}) in}
| \textbf{baseType_{0}.startSet_{0}}
| \textbf{endlet}

| \textbf{<composite state> =>}
| if \textbf{id.s} <composite state structure> \textbf{.s-implicit} \in <composite state body> then
| \{ \textbf{s} <state>: s.parentAS0 = id.s <composite state structure> \textbf{.s-implicit} \}
| else \textbf{sp.stateSet_{0}.sp} <state partition>.sp.parentAS0 = id.s <composite state structure> \textbf{.s-implicit} \end{\textbf{case}}

| \textbf{<typebased composite state> =>}
| \textbf{id.s} <type expression>.baseType_{0}.stateSet_{0}

| \textbf{<composite state reference> =>}
| \textbf{id.referencedDefinition_{0}.stateSet_{0}}

| \textbf{otherwise} \emptyset

\textbf{endcase}

\textbf{localStateSet} \textbf{(id: <agent definition> \cup <agent type definition> \cup <composite state type definition> \cup <procedure definitions>): <state> = def}

\textbf{case id of}

| \textbf{<agent definition> \cup <agent type definition> \cup <composite state type definition> \cup <procedure definitions>} =>

\textbf{endif}

| \textbf{<typebased composite state> => id.s <type expression>.baseType_{0}.startSet_{0}}

| \textbf{<composite state reference> => id.referencedDefinition_{0}.startSet_{0}}

| \textbf{otherwise} \emptyset

\textbf{endcase}
Determine if two operation signatures are compatible.

Auxiliary Functions

not add argument virtuality to any argument of the inherited operation signature.
the parameter kind in any argument of the inherited operation signature. A redefinition of a virtual method must
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 base type, and further, if the
When a method is redefined in a specialization, its signature must be sort compatible with the corresponding signature in

2.5.8.4 Virtual Method

Conditions on Concrete Syntax

∀os ∈ <operation signature> (∃os′ ∈ <operation signature>) ⇒
   (os.entityKind = method) ∧ (os.virtuality ∈ {redefined, finalized})
   ⇒ ∃os′ ∈ <operation signature> : (os′ ∈ os.superCounterparts) ∧
      isOperationSignatureCompatible(os, os′) ∧
      (let fpl = os.operationSignatureParameterList₀ in
       let fpl′ = os′.operationSignatureParameterList₀ in
       ∀i ∈ 1..fpl.length:
       (fpl[i].s <parameter virtuality> = undefined) ⇒
       ((fpl[i].s <argument virtuality> = undefined) ⇒ (fpl′[i].s <argument virtuality> = undefined))
      )

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(os, os′): BOOLEAN =
   isSortCompatible(os.result, os′.result) ∧
   isOperationSignatureCompatibleParameterList₀ in
   let fpl = os.operationSignatureParameterList₀ in
   let fpl′ = os′.operationSignatureParameterList₀ in
   (fpl.length = fpl′.length) ∧
   ∀i ∈ 1..fpl.length:
   (isOperationSignatureCompatibleParameter(fpl[i].s <formal parameter>, fpl′[i].s <formal parameter>) ∧
   (fpl[i].s <argument virtuality> = undefined) ⇒
   (fpl′[i].s <argument virtuality> = undefined) ∧

ITU-T Z.100/Annex F2 (11/2000) 71
((fp[i].s.<argument virtuality>=undefined)⇒
isSameSort(fp[i].s.<formal parameter>.s.<sort>, fp'[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

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

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

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

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

Concrete Syntax

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

<agent structure> ::
  [ <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> ::
<entity in agent> =
    <signal definition>
    | <signal reference>
    | <signal list definition>
    | <variable definition>
    | <remote procedure definition>
    | <remote variable definition>
    | <data definition>
    | <data type reference>
    | <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

∀ sp ∈ <state partition>: sp.parentAS0.parentAS0 ∈ <agent structure> ⇒
sp.name = sp.parentAS0.parentAS0.name0

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

∀ in <initial number>, ∀ mn ∈ <maximum number>:
in.parentAS0 = mn.parentAS0 ⇒
in.s.<simple expression>.value0 ≤ mn.s.<simple expression>.value0 ∧
mn.s.<simple expression>.value0 ≥ 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,
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.

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 $\text{value}_0$ computes the Natural value for a simple expression.

\[
\text{value}_0(e: \text{<simple expression>}) : \text{NAT} = \text{def} \text{value}_1((e))
\]

The function $\text{simpleMapping}$ 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>.

\[
\text{simpleMapping}(e: \text{<simple expression>}) : \text{Constant-expression} = \text{def}
\begin{cases}
\text{case } e \text{ of} \\
| \text{<expression gen primary>(undefined, <operator application>(ident, params))} => \\
\text{mk-Operation-application}(\text{simpleMapping}(\text{ident}), \text{simpleMapping}(\text{x | x in params})) \\
| \text{<expression gen primary>(undefined, <literal identifier>(qual, name))} => \\
\text{mk-Literal-identifier}(\text{simpleMapping}(\text{qual}), \text{simpleMapping}(\text{name})) \\
| \text{<expression gen primary>(undefined, <conditional expression>(e1, e2, e3))} => \\
\text{mk-Conditional-expression}(\text{simpleMapping}(\text{e1}), \text{simpleMapping}(\text{e2}), \text{simpleMapping}(\text{e3})) \\
\text{else} \\
\text{undefined}
\end{cases}
\]

2.6.1 System

Concrete Syntax

\[
<\text{system definition}> ::=
<\text{package use clause}>* <\text{system heading}> <\text{agent structure}>
\]

<\text{system heading}> ::=
<\text{system}<\text{name}> <\text{agent additional heading}>

Conditions on Concrete Syntax

\[
\forall \text{sd} \in <\text{system heading}>: \text{sd.agentLocalFormalParameterList}_0 = \text{empty}
\]

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

Transformations

\[
\begin{align*}
\text{let } nn = \text{newName in} \\
< c =<\text{channel definition}>(n, d, \\
< <\text{channel path}>(ep1=<\text{channel endpoint}(\text{env, undefined}, ep2, list1), \\
\text{undefined}) > \\
\text{provided } c.\text{parentAS0}.\text{parentAS0} = <\text{system definition}>=8=> \\
< <\text{textual gate definition}>(nn), \\
< <\text{channel definition}>(n, d, <\text{channel path}>(<\text{channel endpoint}(\text{env, nn}, ep2, list1), \text{undefined} >)
\end{align*}
\]

\[
\begin{align*}
\text{let } nn = \text{newName in} \\
< c =<\text{channel definition}>(n, d, \\
< <\text{channel path}>(ep1=<\text{channel endpoint}(\text{env, undefined}, ep2, list1), \\
< <\text{channel path}>(\text{ep2, ep1, list2}) ) > \\
\text{provided } c.\text{parentAS0}.\text{parentAS0} = <\text{system definition}>=8=> \\
< <\text{textual gate definition}>(nn), \\
< <\text{channel definition}>(n, d, \\
< <\text{channel path}>(<\text{channel endpoint}(\text{env, nn}, ep2, list1), \\
< <\text{channel path}>(\text{ep2, <channel endpoint>(\text{env, nn}, list2) }) >
\end{align*}
\]

\[
\begin{align*}
\text{let } nn = \text{newName in} \\
< c =<\text{channel definition}>(n, d, \\
< <\text{channel path}>(ep1, ep2=<\text{channel endpoint}(\text{env, undefined}, list1), \\
\text{undefined} >)
\end{align*}
\]
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

```plaintext
<block definition> ::=
 <package use clause>* <block heading> <agent structure>
```

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

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

2.6.4 Procedure

Abstract Syntax

```plaintext
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> } *
Conditions on Concrete Syntax

∀pd∈<procedure definitions>: pd.isExported₀ ⇒
  pd.formalContextParameterList₀ = empty∧
  pd.surroundingScopeUnit₀ ∈ <agent type definition> ∪ <agent definition>

An exported procedure cannot have formal context parameters and its enclosing scope must be an agent type or agent definition.

∀vd∈<variable definition>:
  vd.surroundingScopeUnit₀ ∈ <procedure definitions> ⇒ ¬vd.isExported₀

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

∀pd₁, pd₂∈<procedure definitions>:
  (parentAS₀ofKind(pd₁, <agent definition> ∪ <agent type definition>) = 
  parentAS₀ofKind(pd₂, <agent definition> ∪ <agent type definition>) ∧
  pd₁.isExported₀∧ pd₂.isExported₀∧ pp₁ ≠ pp₂) ⇒
  (pd₁.s.<procedure heading>.s.<procedure preamble>.s.<exported>.s.<identifier> ≠
  pd₁.s.<procedure heading>.s.<procedure preamble>.s.<exported>.s.<identifier>)

Two exported procedures in an agent cannot mention the same <remote procedure identifier>.

∀te∈<type expression>: te.baseType₀ ∉ <external procedure definition> ∧
  ∀procCons∈<procedure constraint>: procCons.s.<identifier> ≠ undefined ⇒
  getEntityDefinition(procCons.s.<identifier>, procedure) ∉
  <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 ≠ undefined ∧
  replaceInSyntaxTree(<return body>(undefined), <return body>(resName), body) ≠ body
  =⇒
  <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 ≠ undefined ∧
  resName ∉ { v.s.<name> | v ∈ <variables of sort gen name>: v.parentAS0.parentAS0.parentAS0 = p }
=8=>
\[\text{\textless procedure definition\textgreater}(uses, h,}
\text{entities \textasciitilde \textless variable definition\textgreater}(\text{undefined, <}
\text{\textless variables of sort\textgreater}(\text{< variables of sort gen name\textgreater}(\text{resName, undefined >, resSort, undefined >),}
\text{body})
\]
A \textless procedure result\textgreater\ with \textless variable name\textgreater\ is derived syntax for a \textless variable definition\textgreater\ with \textless variable name\textgreater\ and
\textless sort\textgreater\ in \textless variables of sort\textgreater. If there is a \textless variable definition\ involving \textless variable name\ no further \textless variable definition\ is added.

\textit{The following statement is covered by the dynamic semantics.}

A \textless procedure start area\textgreater which contains \textless virtuality\textgreater, a procedure \textless start\textgreater which contains \textless virtuality\textgreater, or a \textless statement list\textgreater in a \textless procedure definition\ following \textless virtuality\textgreater is called a virtual procedure start. Virtual procedure start is further described in 8.3.3.

\textless <external procedure definition>(*, *) =7=> empty() \textgreater
An external procedure definition is not considered in the dynamic semantics.

\textbf{Mapping to Abstract Syntax}

\[\text{\textless procedure definition\textgreater}(*,<procedure heading>(*,*,*,,parent,parms,result,*),entities,body)\]
\[\Rightarrow \text{mk-Procedure-definition(Mapping(name), Mapping(result), Mapping(parent),}
\{ e \in Mapping(entities).toSet: (e \in Data-type-definition) \},
\{ e \in Mapping(entities).toSet: (e \in Syntype-definition) \},
\{ e \in Mapping(entities).toSet: (e \in Variable-definition) \},
\{ e \in Mapping(entities).toSet: (e \in Composite-state-type-definition) \},
\{ e \in Mapping(entities).toSet: (e \in Procedure-definition) \},
\text{Mapping(body)})\]

\[\text{\textless procedure definition gen compound statement\textgreater}(*, body) \Rightarrow \text{Mapping(body)}\]

\[\text{\textless formal procedure parameter\textgreater}(\text{in, params}) \Rightarrow \text{mk-In-parameter(Mapping(params))}\]

\[\text{\textless formal procedure parameter\textgreater}(\text{out, params}) \Rightarrow \text{mk-Out-parameter(Mapping(params))}\]

\[\text{\textless formal procedure parameter\textgreater}(\text{inout, params}) \Rightarrow \text{mk-Inout-parameter(Mapping(params))}\]

\textbf{Auxiliary Functions}

Determine if a \textless variable definition\ or a \textless procedure definition\ is exported.

\textit{isExported}(vp: <variable definition>\textasciitilde\textless procedure definition\textgreater\): BOOLEAN=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 \textless procedure definition\.

\textit{procedureFormalParameterList}(pd: <procedure definition>\): <name>*=def
\textit{pd.localProcedureFormalParameterList0} \cap \textit{pd.inheritedProcedureFormalParameterList0}

\textit{localProcedureFormalParameterList}(pd: <procedure definition>\): <name>*=def
let fpl=pd.s-<procedure heading>.s-<formal procedure parameter>-seq in
  if fpl = empty then empty
  else
    \textless f.s-parameters of sort\textgreater.s-<variable>-seq | f in fpl >
  endif
endlet
inheritedProcedureFormalParameterList(pd: <procedure definition>): <name>* =def
  let sp = pd.specialization0 in
  if sp = undefined then empty
  else sp.s.<type expression>.baseType0.procedureFormalParameterList0
endlet

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

procedureSignatureParameterList(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(d: Agent-type-definition ∪ Composite-state-type-definition ∪ Procedure-definition):
  Parameter* =def
  d.localFormalParameterList1∩d.inheritedFormalParameterList1
localFormalParameterList(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
endlet

Determine if the sort list of Expressions corresponds by position to the list of Sort-reference-identifiers.

isActualAndFormalParameterMatched([expl: [Expression]*, fpsl: Sort-reference-identifier*], BOOLEAN =edef
  (expl.length = fpsl.length)∧
  (∀i∈ 1..expl.length:expl[i]≠undefined⇒isCompatibleTo(expr[i],staticSort,Fpsl[i]))

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

  case d of
    | Agent-type-definition ∪ Composite-state-type-definition ∪ Procedure-definition =>
      <param.s.Sort-reference-identifier | param in d.formalParameterList1>
    | Operation-signature => <fa.s.Argument | fa in d.s.Formal-argument-seq>
    | Signal-definition ∪ Timer-definition ∪ Exception-definition => d.s.Sort-reference-identifier-seq
  endcase

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

isSameRaises(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

∀c ∈ Channel-definition: |c.s-Channel-path-set| ∈ {1, 2}

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

∀c ∈ Channel-definition: |c.s-Channel-path| = 2 ⇒

∀p, p' ∈ c.s-Channel-path: p ≠ p' ⇒

p.s-Originating-gate = p'.s-Destination-gate ∧ p'.s-Originating-gate = p.s-Destination-gate

When there are two paths, the channel is bidirectional and the Originating-gate of each Channel-path must be the same as the Destination-gate of the other Channel-path.

∀c ∈ Channel-definition: ∀p ∈ Channel-path: ∀d ∈ Agent-type-definition:

∀g, g' ∈ Gate-definition: (p.parentASI = c) ∧ (g, parentASI = d) ∧ (g', parentASI = d) ∧ (g ≠ g') ∧

(p.s-Originating-gate = g.s-identifier₁) ∧ (p.s-Destination-gate = g'.s-identifier₁) ⇒ |c.s-Channel-path-set| = 1

If the Originating-gate and the Destination-gate are in the same Agent-definition, the channel must be unidirectional (there must be only one element in the Channel-path-set).

∀c ∈ Channel-definition: ∀p ∈ Channel-path: ∃g, g' ∈ Gate-definition:

(p.parentASI = c) ∧ (p.s-Originating-gate = g.s-identifier₁) ∧ (g, parentASI = c. parentASI) ∧

(p.s-Destination-gate = g'.s-identifier₁) ∧ (g'. parentASI = c. parentASI)

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

∀c ∈ <channel endpoint>:

let id = ce.s-implicit in
let g = ce.s-gate in

(id ∈ <identifier> ∧ getEntityDefinition(id, agent) ∈ <textual type-based agent definition>) ⇒

(let d = getEntityDefinition(id, agent) in

let id = d.s-<type expression>.baseType₀ in

let id = ce.s-<gate> # undefined ∧

(∃gd ∈ <textual gate definition>: ∃ge ∈ <gate constraint>);
<gate> must be specified if <channel endpoint> denotes a connection to a <textual type-based 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.

\[ \forall ce \in <\text{channel endpoint}> : \]

\[
\begin{align*}
& \text{let } id = ce.s-\text{implicit in} \\
& \text{let } g = ce.s-<\text{gate} \text{ in} \\
& \quad (id \in <\text{identifier}>) \land \\
& \quad \text{getEntityDefinition}(id, state) \in <\text{textual type-based state partition definition}>) \Rightarrow \\
& \quad (let d = getEntityDefinition(id, state) in \\
& \quad \text{let } td = d.s-<\text{type expression}>.baseType_0 \text{ in} \\
& \quad (ce.s-<\text{gate} \neq \text{undefined} \land \\
& \quad (\exists gd \in <\text{textual gate definition}>: \exists ge \in <\text{gate constraint}>: \\
& \quad g \in td.localGateDefinitionSets_0 \land gc.parentAS_0 = gd \land \\
& \quad \text{ges} \in <\text{signal list item}>.seq.signalSets_0 \land ce.s-<\text{signal list item}>.seq.signalSets_0 \neq 0) \\
& \end{align*}
\]

endlet)

\[ \text{endlet} \]

<gate> must be specified if <channel endpoint> denotes a connection to a <textual type-based 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.

\[ \forall ce \in <\text{channel endpoint}> : \]

\[
\begin{align*}
& \text{let } id = ce.s-\text{implicit in} \\
& \text{let } g = ce.s-<\text{gate} \text{ in} \\
& \quad (id \in <\text{identifier}>) \land \\
& \quad \text{getEntityDefinition}(id, state) \in <\text{textual type-based state partition definition}>) \Rightarrow \\
& \quad (let d = getEntityDefinition(id, state) in \\
& \quad \text{let } td = d.s-<\text{type expression}>.baseType_0 \text{ in} \\
& \quad (ce.s-<\text{gate} \neq \text{undefined} \land \\
& \quad (\exists gd \in <\text{textual gate definition}>: \exists ge \in <\text{gate constraint}>: \\
& \quad g \in td.localGateDefinitionSets_0 \land gc.parentAS_0 = gd \land \\
& \quad \text{ges} \in <\text{signal list item}>.seq.signalSets_0 \land ce.s-<\text{signal list item}>.seq.signalSets_0 \neq 0) \\
& \end{align*}
\]

endlet)

\[ \text{endlet} \]

<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.

\[ \forall ce \in <\text{channel endpoint}> : \]

\[
\begin{align*}
& \text{let } id = ce.s-\text{implicit in} \\
& \text{let } g = ce.s-<\text{gate} \text{ in} \\
& \quad (id \in <\text{identifier}>) \land \\
& \quad \text{getEntityDefinition}(id, state) \in <\text{agent type definition}>) \Rightarrow \\
& \quad (let d = getEntityDefinition(id, state) in \\
& \quad \text{let } td = d.s-<\text{type expression}>.baseType_0 \text{ in} \\
& \quad (ce.s-<\text{gate} \neq \text{undefined} \land \\
& \quad (\exists gd \in <\text{textual gate definition}>: \exists ge \in <\text{gate constraint}>: \\
& \quad g \in td.localGateDefinitionSets_0 \land gc.parentAS_0 = gd \land \\
& \quad \text{ges} \in <\text{signal list item}>.seq.signalSets_0 \land ce.s-<\text{signal list item}>.seq.signalSets_0 \neq 0) \\
& \end{align*}
\]

endlet)

\[ \text{endlet} \]

Transformations

\[ <\text{channel definition}> (\text{undefined}, \text{delay}, p1, p2) = 1 \Rightarrow \]

\[ <\text{channel definition}> (<\text{name}, \text{delay}, p1, p2) \]

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

\[ t = <\text{textual type-based agent definition}> \]

\[ \text{provided } \text{unconnectedGates}(t) = \text{undefined} \]

\[ = 9 \Rightarrow t \]

and

\[ \text{unconnectedGates}(t) := \]

\[
\begin{align*}
& (let id = \text{take}(| \text{te.s-<base type>} | \text{te} \in <\text{type expression}>: \text{te.parentAS_0.parentAS} = t |) \text{ in} \\
& \quad (g \in \text{id.refersto0.getEntities.toSet:} \\
& \quad g \in <\text{gate in definition}> \land \neg \text{isConnected}(g, \text{undefined}, \text{id.refersto0.getEntities.toSet} ) \\
& \end{align*}
\]

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):

\[ t = \text{<agent type definition>} \]

\[ \text{provided } \text{unconnectedGates}(t) = \text{undefined} \]

\[ \text{and} \]

\[ \text{unconnectedGates}(t) := \{ \text{g } \in \text{t.getEntities.toSet: g } \in \text{gate in definition } \land \text{isConnected(g, undefined, t.getEntities.toSet) } \} \]

\[ \text{<a1} = \text{<textual typebased agent definition>} > \text{^ something} > \text{<a2} = \text{<textual typebased agent definition>} > 

\[ \text{provided } \text{missingConnections(a1,a2)} \neq \text{∅} \]

\[ \Rightarrow \text{let c = <channel definition>(newName, undefined, missingConnections(a1,a2).take, undefined) in} \]

\[ \text{<a1} > \text{^ something} \text{<a2} > \text{^ c} \]

\[ \text{endlet} \]

\[ \text{<a1} = \text{<textual typebased agent definition>} > \text{^ something} \text{<a2} = \text{<textual typebased agent definition>} > 

\[ \text{provided } \text{missingConnections(a2,a1)} \neq \text{∅} \]

\[ \Rightarrow \text{let c = <channel definition>(newName, undefined, missingConnections(a2,a1).take, undefined) in} \]

\[ \text{<a1} > \text{^ something} \text{<a2} > \text{^ c} \]

\[ \text{endlet} \]

\[ \text{<a} = \text{<textual typebased agent definition>} > 

\[ \text{provided } \text{missingConnections(a,a.parentAS0.parentAS0)} \neq \text{∅} \]

\[ \Rightarrow \text{let c = <channel definition>(newName, undefined, missingConnections(a,a.parentAS0.parentAS0).take, undefined) in} \]

\[ \text{<a} > \text{^ c} \]

\[ \text{endlet} \]

\[ \text{<a} = \text{<textual typebased agent definition>} > 

\[ \text{provided } \text{missingConnections(a.parentAS0.parentAS0,a)} \neq \text{∅} \]

\[ \Rightarrow \text{let c = <channel definition>(newName, undefined, missingConnections(a.parentAS0.parentAS0,a).take, undefined) in} \]

\[ \text{<a} > \text{^ c} \]

\[ \text{endlet} \]
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 type based 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)
\[ \text{mk-Channel-path} (\text{Mapping}(\text{endp1}), \text{Mapping}(\text{endp2}), \text{Mapping}(\text{with})) \]

| <channel endpoint>(\text{*}, \text{gate}) \Rightarrow \text{Mapping}(\text{gate}) |

**Auxiliary Functions**

\[ \text{SIGNALDIRECTION}_0 \text{=} \text{def} \{ \text{out}, \text{in} \} \]

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

\[ \text{direction}(p: \text{<gate constraint> } \cup \text{<channel endpoint>}) : \text{SIGNALDIRECTION}_0 \text{=} \text{def} \]

\[
\begin{align*}
\text{case } p \text{ of } & \\
| \text{<channel endpoint> } & \Rightarrow \text{if } p = p.\text{parentAS0}. s - \text{<channel endpoint> } \text{ then out else in endif } \\
| \text{<gate constraint> } & \Rightarrow p.\text{s-implicit} \\
\text{otherwise undefined} \\
\end{align*}
\]

endcase

The function \text{origination}_0 gets the originating channel endpoint of a <channel path>.

\[ \text{origination}_0(p: \text{<channel path>}) : \text{<channel endpoint>} = \text{def} \]

\[ p.\text{s-}\text{<channel endpoint>} \]

The function \text{destination}_0 gets the destination channel endpoint of a <channel path>.

\[ \text{destination}_0(p: \text{<channel path>}) : \text{<channel endpoint>} = \text{def} \]

\[ p.\text{s2-}\text{<channel endpoint>} \]

The function \text{channelEndpointReferTo}_0 is used to get the entity definition that the <channel endpoint> referred to.

\[ \text{channelEndpointReferTo}_0(ep: \text{<channel endpoint>}) : \text{ENTITYDEFINITION}_0 \text{=} \text{def} \]

\[
\begin{align*}
\text{let } \text{end } = ep.\text{s-implicit in} & \\
\text{case } \text{end } & \text{ of} & \\
| \text{<identifier> } & \Rightarrow & \text{getEntityDefinition}_0(\text{end}, \text{end}.\text{idKind}_0) & \\
| \text{this } & \Rightarrow & \text{parentAS0ofKind}(\text{end}, \text{<agent definition> } \cup \text{<agent type definition> } ) & \\
| \text{env } & \Rightarrow & \text{undefined} & \\
\text{endcase} & \\
\text{endlet} & \\
\end{align*}
\]

The function \text{unconnectedGates} is used to store the gates that are not explicitly connected.

\[ \text{controlled unconnectedGates} : \langle\text{textual typebased agent definition} \rangle \cup \langle\text{agent type definition} \rangle \rightarrow \langle\text{gate in definition} \rangle \text{-set} \]

The function \text{missingConnections} is used to compute the missing implicit connections between two agents.

\[ \text{missingConnections}(ag1: \langle\text{textual typebased agent definition} \rangle \cup \langle\text{agent type definition} \rangle, \\
ag2: \langle\text{textual typebased agent definition} \rangle \cup \langle\text{agent type definition} \rangle) : \langle\text{channel path} \rangle \text{-set } = \text{def} \]

\[
\begin{align*}
\text{let } \text{entities } & = & \\
\text{if } ag1 \in \langle\text{agent type definition} \rangle & \text{ then } ag1.\text{getEntities} & \\
\text{elseif } ag2 \in \langle\text{agent type definition} \rangle & \text{ then } ag2.\text{getEntities} & \\
\text{else } \text{parentAS0ofKind}(ag1, \langle\text{agent type definition} \rangle).\text{getEntities} & \text{ endif} & \\
\text{in} & \\
\text{let } \text{id1 } & = & \\
\text{if } ag1 \in \langle\text{agent type definition} \rangle & \text{ then } \text{env else ag1.identifier}_0 & \text{ endif} & \\
\end{align*}
\]
The function `isConnected` is used to check whether two gates are connected.

```plaintext
isConnected(g1: <gate in definition>, g2: <gate in definition>, ent: DefinitionAS0-set): BOOLEAN =)
  let allPathes =
    U { { e.s-<channel path> } ⋃
      if e.s2-<channel path> = undefined then ∅ else { e.s2-<channel path> } endif
    | e ∈ ent.toSet: e ∈ <channel definition> } in
  ∃ p ∈ allPathes: g1 = p.s-<channel endpoint>s-<gate>.refersto0 ∧
    (g2 = p.s2-<channel endpoint>s-<gate>.refersto0 ∨ g2 = undefined)
endlet
```

The function `inoutSignals` is used to compute the outwards or inwards going signals of a gate.

```plaintext
inoutSignals(g: <textual gate definition>, kind: { in, out }): <signal list item>-set =)
  U { g.s-<signal list item>-seq | g ∈ <gate constraint>: g.s-implicit = kind }
```

### 2.7.2 Connections

#### Concrete Syntax

```plaintext
<channel to channel connection> ::
  <channel><identifier>+ <channel><identifier>+
```

#### Conditions on Concrete Syntax

∀c1,c2∈<channel to channel connection>:

```plaintext
(let ids1 = c1.s2-<identifier>-seq.toSet in
let ids2 = c2.s2-<identifier>-seq.toSet in
  c1.surroundingScopeUnit0 = c2.surroundingScopeUnit0 ∧ c1 ≠ c2 ⇒ ids1 ∩ ids2 = ∅)
endlet
```

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>:

```plaintext
(let ids1 = c1.s-<identifier>-seq.toSet in
let ids2 = c2.s-<identifier>-seq.toSet in
  (c1.surroundingScopeUnit0 = c2.surroundingScopeUnit0 ∧ c1 ≠ c2) ⇒ (ids1 = ids2 ∨ ids1 ∩ ids2 = ∅)
endlet
```

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

```plaintext
let nn = newName(undefined) in
< c=<channel to channel connection>(*, *): provided c.myImplicitGateIdentifier = undefined
  =⇒< c, <textual gate definition>(nn, <gate constraint>(out, allSignalsOut(c)), <gate constraint>(in, allSignalsIn(c)) ) >
and
  c.myImplicitGateIdentifier = <identifier>(fullQualifier(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.

\[
c = \langle \text{channel endpoint} \rangle (id, \text{undefined}) \quad \text{provided} \quad \text{findconnect}(c.\text{parentAS0}, \text{parentAS0}, id) \neq \text{undefined}
\]

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.

\[
\text{controlled myImplicitGateIdentifier}: \langle \text{channel to channel connection} \rangle \rightarrow \langle \text{identifier} \rangle
\]

The function findconnect computes the implicit gate identifier for a channel that is mentioned in a channel-to-channel connection.

\[
\text{findconnect}(ch: \langle \text{channel definition} \rangle, id: \text{DefinitionAS0}): \langle \text{identifier} \rangle = \text{def}
\]

\[
\text{if id} = \text{env then}
\]

\[
\begin{align*}
\text{let matchingGateIds} &= \{ c.\text{myImplicitGateIdentifier} | c \in \langle \text{channel-to-channel connection} \rangle : \\
& c.\text{parentAS0} = ch.\text{parentAS0} \land \text{fullIdentifier}(c) \in c.\text{s2-<identifier>-seq} \} \text{ in} \\
& \text{matchingGateIds}.\text{take}
\end{align*}
\]

\[
\text{else}
\]

\[
\begin{align*}
\text{let matchingGateIds} &= \{ c.\text{myImplicitGateIdentifier} | c \in \langle \text{channel-to-channel connection} \rangle : \\
& c.\text{parentAS0} = id.\text{refersto0} \land \text{fullIdentifier}(c) \in c.\text{s-<identifier>-seq} \} \text{ in} \\
& \text{matchingGateIds}.\text{take}
\end{align*}
\]

\[
\text{endif}
\]

The function allSignalsIn computes the input signals belonging to a channel-to-channel connection.

\[
\text{allSignalsIn}(c: \langle \text{channel to channel connection} \rangle): \text{DefinitionAS0}* = \text{def}
\]

\[
\text{BigSeq}( < id.\text{refersto0}.s.<\text{channel path}>.s.<\text{signal list item}>-seq | id \text{ in c.s-<identifier>-seq} > ) \cap \\
\text{BigSeq}( < id.\text{refersto0}.s.<\text{channel path}>.s.<\text{signal list item}>-seq | id \text{ in c.s2-<identifier>-seq} > )
\]

The function allSignalsOut computes the output signals belonging to a channel-to-channel connection.

\[
\text{allSignalsOut}(c: \langle \text{channel to channel connection} \rangle): \text{DefinitionAS0}* = \text{def}
\]

\[
\text{BigSeq}( < id.\text{refersto0}.s.<\text{channel path}>.s.<\text{signal list item}>-seq | id \text{ in c.s-<identifier>-seq} > ) \cap \\
\text{BigSeq}( < id.\text{refersto0}.s.<\text{channel path}>.s.<\text{signal list item}>-seq | id \text{ in c.s2-<identifier>-seq} > )
\]

**Mapping**

\[
| \langle \text{channel to channel connection} \rangle(*,*) | \Rightarrow \text{empty}
\]

### 2.7.3 Signal

**Abstract Syntax**

\[
\text{Signal-definition} :: \text{Signal-name Sort-reference-identifier*}
\]

**Concrete Syntax**

\[
\text{<signal definition>}:: \langle \text{type preamble} \rangle \langle \text{signal definition item} \rangle+
\]

\[
\text{<signal definition item>} ::
\]

\[
\begin{align*}
\langle \text{signal}\langle \text{name} \rangle \langle \text{formal context parameter}* \rangle \langle [\text{virtuality constraint}] \rangle \langle [\text{specialization}] \rangle \langle \text{sort}* \rangle
\end{align*}
\]
Conditions on Concrete Syntax

\[ \forall sdi \in <\text{signal definition item}>: \forall fcp \in <\text{formal context parameter}>:\]

\[ fcp \in sdi.\text{localFormalContextParameterList}_0.\text{toList} \Rightarrow fcp \in <\text{sort context parameter}> \]

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

\[ \forall sdi \in <\text{signal definition item}>: sdi.\text{specialization}_0.s-<\text{type expression}>.s-<\text{base type}>.idKind_0 = \text{signal} \]

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

\[ \forall sid \in <\text{identifier}>: sid.\text{parentAS0} \notin <\text{type expression} > \land sid.\text{parentAS0} \notin <\text{signal constraint}> = > \neg \text{getEntityDefinition}(sid, \text{signal}).\text{isAbstractType}0 \]

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

Transformations

\[ < <\text{signal definition}> (pre, <item > ^ rest) > \text{provided rest \# undefined} = 1 => < <\text{signal definition}> (pre, <item >), <\text{signal definition}> (pre, rest) > \]

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

\[ | <\text{signal definition}> (*, <item>) => \text{Mapping}(item) \]
\[ | <\text{signal definition} item> (name,*,*,*,sortlist) => \text{mk-Signal-definition}(\text{Mapping}(name), \text{if sortlist= undefined then empty else Mapping(sortlist) endif}) \]

2.7.4 Signal List Definition

Concrete Syntax

\[ <\text{signal list definition}> ::=
<\text{signal list name}> <\text{signal list item}>^+ \]

\[ <\text{signal list item}> ::=
[ signal | signal list | timer | remote procedure | interface | remote variable ] <\text{identifier}> \]

Conditions on Concrete Syntax

\[ \forall siglistDef \in <\text{signal list definition}>: \sim \text{isSiglistContaining}(siglistDef, siglistDef) \]

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

\[ < <\text{signal list item}> (kind, id) > \text{provided id.refersto0} \in <\text{signal list definition}> = 8 => id.refersto0,s-<\text{signal list item}>- \text{seq} \]

Every <signal list identifier> is replaced by the list of signals of its definition.

Mapping to Abstract Syntax

\[ | < <\text{signal list definition}> (*,*) => \text{empty} \]
\[ | < <\text{signal list item}> (*, id) => \text{Mapping}(id) \]
Auxiliary Functions

The function \( \text{isSiglistContaining}_0 \) is used to determine if a signal list contains another signal list, either directly or indirectly.

\[
\text{isSiglistContaining}_0(sld1, sld2) = \begin{cases} 
\text{TRUE} & \text{if } \exists \text{ sid } \in \text{ sld1_.signalList }: \text{ sid } = \text{ sld2}\text{.signalList} \text{ and } \text{ sid } = \text{ sld1_.signalList}, \\
\text{FALSE} & \text{otherwise}.
\end{cases}
\]

### 2.7.5 Remote Procedures

#### Concrete Syntax

\[
\text{<remote procedure definition>} ::= \langle\text{remote procedure}\rangle \text{name} \begin{cases} \langle\text{remote procedure}\rangle \text{name}[\text{nodeDelay}] \langle\text{procedure signature}\rangle \\
\langle\text{remote procedure}\rangle \text{name} \end{cases}
\]

\[
\text{<remote procedure call>} ::= \langle\text{remote procedure}\rangle \text{name} \langle\text{actual parameter}\rangle^n \langle\text{communication constraint}\rangle^n \langle\text{on exception}\rangle
\]

#### Conditions on Concrete Syntax

\[
\forall \text{pd} \in \langle\text{procedure definitions}\rangle : \\
\text{let } \text{rpi} = \text{pd.s-<procedure heading>}, \text{s-<procedure preamble>}, \text{s-<exported>}, \text{s-<identifier> in} \\
\text{let } \text{rpd} = \text{getEntityDefinition}_0(\text{rpi, remote procedure}) \text{ in} \\
\text{pd.isExported}_0 \land \text{rpi} = \text{undefined} \Rightarrow \\
\text{isSameProcedureAndSignature}_0(\text{pd, rpd.s-<procedure signature>}) \\
\end{let}
\]

The \langle\text{remote procedure}\rangle identifier following as in an exported procedure definition must denote a \langle\text{remote procedure}\rangle with the same signature as the exported procedure.

\[
\forall \text{pd} \in \langle\text{procedure definitions}\rangle : \\
\text{let } \text{rpi} = \text{pd.s-<procedure heading>}, \text{s-<procedure preamble>}, \text{s-<exported>}, \text{s-<identifier> in} \\
\text{let } \text{rpd} = \text{getEntityDefinition}_0(\text{pd.name}_0, \text{remote procedure}) \text{ in} \\
\text{pd.isExported}_0 \land \text{rpi} = \text{undefined} \Rightarrow \\
(\text{rpd} \neq \text{undefined} \land \text{isSameProcedureAndSignature}_0(\text{pd, rpd.s-<procedure signature>})) \\
\end{let}
\]

In an exported procedure definition with no as clause, the name of the exported procedure is implied and the \langle\text{remote procedure}\rangle definition in the nearest surrounding scope with same name is implied.

\[
\forall \text{rpc} \in \langle\text{remote procedure call}\rangle : \\
(\text{let } \text{d} = \text{parentAS0ofKind}(\text{rpc, <agent definition> } \cup <\text{agent type definition}>) \text{ in} \\
\text{rpc.s-<identifier> in d.validOutputSignalSet}_0) \\
\end{let}
\]

A remote procedure mentioned in a \langle\text{remote procedure call}\rangle must be in the complete output set of an enclosing agent type or agent set.

\[
\forall \text{expr} \in \langle\text{expression}\rangle : \forall \text{callBody} \in \langle\text{remote procedure call body}\rangle : \\
\text{expr, parentAS0, parentAS0} \in \text{callBody} \land \text{expr.staticSort}_0 \neq \text{"Pid"} \Rightarrow \\
(\text{let } \text{def} = \text{getEntityDefinition}_0(\text{expr.staticSort}_0, \text{sort}) \text{ in} \\
\text{let } \text{pd} = \text{getEntityDefinition}_0(\text{callBody.s-<identifier>}, \text{remote procedure}) \text{ in} \\
\text{def} \in \langle\text{interface definition}\rangle \land \text{isDefinedIn}_0(\text{pd, def}) \\
\end{let}
\]

If \langle\text{destination}\rangle in a \langle\text{remote procedure call body}\rangle is a \langle\text{pid}\rangle expression with a sort other than Pid, then the \langle\text{remote procedure}\rangle identifier must represent a remote procedure contained in the interface that defined the pid sort.

\[
\forall \text{id} \in \langle\text{identifier}\rangle : \\
id \in \langle\text{communication constraint}\rangle \Rightarrow \text{getEntityDefinition}_0(\text{id, exception}) = \text{undefined}
\]
The \textit{identifier} of \textit{communication constraints} must not have the same \textit{identifier} as an \textit{exception identifier}.

\[ \forall c \in \text{output body} : \]
\[ (~\forall d1, d2 \in \text{destination} : d1.\text{parentAS0} = d2.\text{parentAS0} \land c = d1.\text{parentAS0} \Rightarrow d1 = d2) \land \]
\[ (~\forall d1, d2 \in \text{identifier} : d1.\text{parentAS0} = d2.\text{parentAS0} \land c = d1.\text{parentAS0} \Rightarrow d1 = d2) \]

A \textit{communication constraints} shall contain no more than one \textit{destination} and no more than one \textit{timer identifier}.

Transformations

A remote procedure call

\begin{verbatim}
call Proc(apar) to destination timer timerlist via viapath
\end{verbatim}

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 \textit{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 \textit{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.

\begin{verbatim}
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> \cap "RAISE", empty, undefined, undefined,
  exREFERsto.s.<sort>\cap seq \cap "Integer")) |
  ex in sign.s.<raises> > in
let eh = <exception handler>(< nn \cap "EXC" >, undefined,
< <handle>(undefined, < <exception stimulus>(ex, params) >, undefined,
<transition gen action statement>(<
  <output body>(<output body gen identifier>(
    <identifier>(ex.s.<qualifier>, ex.s.<name> \cap "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 \cap "CALL", empty, undefined, undefined, < "Integer" >) >),
  <signal definition>({undefined,
    < <signal definition item>(nn \cap "REPLY", empty, undefined, undefined, < "Integer" >) >) >
  \cap eigs \cap < eh >
endlet)
and
r.implicitName := nn
\end{verbatim}

There are two implicit \textit{signal definition}s for each \textit{remote procedure definition}s in a \textit{system definition}. The \textit{signal name}s in these \textit{signal definition}s are denoted by \textit{pCALL} and \textit{pREPLY} respectively, where \textit{p} is uniquely determined. The signals are defined in the same scope unit as the \textit{remote procedure definition}. Both \textit{pCALL} and \textit{pREPLY} have a first parameter of the predefined Integer sort.

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

For every exception \textit{e} contained in the \textit{raises} of the remote procedure, and for every predefined exception the following is inserted:

\begin{verbatim}
exceptionhandler pEXC;
handle e(params);
\end{verbatim}
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.refersto0 = 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.refersto0.implicitName ≠ undefined

=17=>
< <channel definition>(n, delay,
<channel path>(ep1, ep2, sigs1 ∩ <identifier>(q, i.refersto0.implicitName ∪ “CALL”) > ∩ sigs2),
path2, n2),
<channel definition>(newName,
<channel path>(ep2, ep1, <identifier>(q, i.refersto0.implicitName ∪ “REPLY”)),
undefined, undefined)>

On each channel mentioning the remote procedure, the remote procedure is replaced by pCALL. For each such channel, a new channel is added in the opposite direction; this channel carries the signal pREPLY. The new channel has the same delaying property as the original one.

let nn=newName in
let varN = nn ∪ “N” in
let varNewN = nn ∪ “NewN” in
r=<remote procedure call>(<remote procedure call body>(id, params, constr))
=17=> <procedure call>(<procedure call body>(undefined, <identifier>(undefined, nn), empty))

and

let varDefs = <
<variable definition>(undefined, <
<variables of sort>(<variables of sort gen name>(varN, undefined) >, “Integer”, “1”) >),
<variable definition>(undefined, <
<variables of sort>(<variables of sort gen name>(varNewN, undefined) >, “Integer”, undefined) >)
>

in
let timerInput = <
<input part>(undefined,
<stimulus>(tid, params) >,
undefined, 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>(<
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.refersto0 ∈ <remote procedure definition>
=17=> <input part>(rpc.refersto0,implicitName ^ “CALL”, fpar ^ n, <transition>(<
<task>(<assignment>(ivar, <expression gen primary>(undefined, <sender expression>(())), undefined),
if rpc.refersto0.s.<procedure signature>.s.<result> ≠ undefined then
<procedure call>(<procedure call body>(undefined, rpc, fpar), rpc.refersto0,implicitName ^ “EXC”)
else
<task>(<assignment>(res,
  <value returning procedure call>(<procedure call body>(undefined, rpc, fpar))),
  rpc.refersto0,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.name0, states) then s
  else <state>(s.s.<state list>, s.s.<on exception>, s.s.-implicit ^
  <input part>(rpc.refersto0,implicitName ^ “CALL”, fpar ^ n, <transition>(<
  <task>(<assignment>(ivar,
    <expression gen primary>(undefined, <sender expression>(())), undefined),
  if rpc.refersto0.s.<procedure signature>.s.<result> ≠ undefined then
    <procedure call>(
      <procedure call body>(undefined, rpc, fpar), rpc.refersto0,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): ENTITYDEFINITION0* =def`

```plaintext
if n = undefined then undefined else
  case n of
  | <agent type definition> => n.s.<agent structure>.s.<entity in agent>-seq
  | <agent definition> => n.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(pd: <procedure definition>, ps: <procedure signature>): BOOLEAN =def`

```plaintext
let fpl = ps.procedureSignatureParameterList0 in
let fpl' = pd.procedureFormalParameterList0 in
(fpl.length = fpl'.length) ∧
(∀ i ∈ 1..fpl.length:
  (fpl[i].s.<parameter kind> = fpl'[i].parentAS0.parentAS0.s.<parameter kind>) ∧
  isSameSort(fpl[i].s.<sort>, fpl'[i].parentAS0.s.<sort>)) ∧
isSameResult(pd.s.<procedure heading>.s.<procedure result>, ps.s.<result>) ∧
isSameRaises(pd.s.<procedure heading>.s.<raises>, ps.s.<raises>)
endlet
```
Determine if two results are matching.

\[
\text{isSameResult}(r, r') \equiv (r \cup \text{<procedure result>} \cup \text{<operation result>}, r' \cup \text{<procedure result>} \cup \text{<operation result>}) = \text{BOOLEAN}
\]

\[
\text{isSameSort}(r.s, r'.s) \equiv (r.s \neq \text{undefined} = \text{isSameSort}(r'.s, r.s))
\]

### 2.7.6 Remote Variables

#### Concrete Syntax

\[
\text{<remote variable definition>} ::= \text{<remote variable definition gen name>}+ \\
\text{<remote variable definition gen name>} ::= \text{<remote variable}<name> + \text{<sort>} [\text{nodelay}] \\
\text{<import expression>} ::= \text{<remote variable}<identifier> \text{<communication constraint>}* \\
\text{<export>} ::= \text{<export body>} [<\text{on exception}>] \\
\text{<export body>} ::= \text{<variable}<identifier>+ \\
\]

#### Conditions on Concrete Syntax

\[
\forall v \in \text{<variables of sort gen name>}:\
\text{let } rvd = \text{getEntityDefinition}(v.s.<identifier>, \text{remote variable}) \text{ in} \\
v.isExported0 \land v.s.<identifier> \neq \text{undefined} = \text{isSameSort}(v.parentAS0.s.<sort>, rvd.s.<sort>)
\]

The \text{<remote variable identifier>} following as in an exported variable definition must denote a \text{<remote variable definition>} of the same sort as the exported variable definition.

\[
\forall v \in \text{<variables of sort gen name>}:\
\text{let } rvd = \text{getEntityDefinition}(v.s.<name>, \text{remote variable}) \text{ in} \\
v.isExported0 \land v.s.<identifier> = \text{undefined} \Rightarrow (rvd = \text{undefined} \land \text{isSameSort}(v.parentAS0.s.<sort>, rvd.s.<sort>))
\]

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.

\[
\forall exps <\text{import expression}> : exp.s.<identifier> \in \\
(\text{let } d = \text{parentAS0ofKind}(exp, \text{<agent definition> } \cup \text{<agent type definition>}) \text{ in} \\
exp.s.<identifier> \in d.\text{validOutputSignalSet}_0) \\
\text{endlet}
\]

A remote variable mentioned in an \text{<import expression>} must be in the complete output set of an enclosing agent type or agent set.

\[
\forall vid \in \text{<identifier>} : vid.parentAS0 \in <\text{export body}> \Rightarrow \text{getEntityDefinition}(vid, \text{variable}).isExported0
\]

The \text{<variable identifier>} in \text{<export>} must denote a variable defined with exported.

\[
\forall exps <\text{expression}> : \forall importExp <\text{import expression}> : \\
\text{exp. parentAS0} = \text{importExp} \land \text{exp.staticSort0} \neq \text{"Pid"} \Rightarrow \\
(\text{let } def = \text{getEntityDefinition}(exp.staticSort0, \text{sort}) \text{ in} \\
\text{let } pd = \text{getEntityDefinition}(importExp.s.<identifier>, \text{remote variable}) \text{ in} \\
defin \in <\text{interface definition} > \land \text{isDefinedIn}(pd, def) \\
\text{endlet})
\]

If \text{<destination>} in an \text{<import expression>} is a \text{<pid expression>} with a sort other than Pid, then the \text{<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.refersto0.implicitName ≠ undefined
     =16=>
       < <channel definition>(n, delay,
       <channel path>(ep1, ep2, sigs1 ∩ < <identifier>(q, i-s<name> ∩ “QUERY” ) > ∩ sigs2),
       path2, n2),
       <channel definition>(newName,
       <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 

<x exception definition>(<x 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>(<x variables of sort gen name>(varN, undefined) ▽ “Integer”, “1”)
      >,
      <variable definition>(undefined, <
         <variables of sort>(<x 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>(<operator application>(“=” , < varNewN, varN >),
   <answer>(“True”, empty))

For each imported variable, two implicit Integer variables \( n \) and \( \text{newn} \) are defined, and \( n \) is initialized to 0. In addition, an implicit variable \( x \) of the sort of the remote variable is defined.

The \(<\text{import expression}>\)

\[
\text{import } (x, \text{destination via via-path})
\]

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:

\[
\begin{align*}
\text{task } & \ n := n + 1; \\
\text{output } & \ x\text{QUERY}(n) \text{ to destination via via-path; }
\end{align*}
\]

wait in state xWAIT, saving all other signals;

input xREPLY(x,\text{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 \(<\text{communication constraints}>\), 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.refersto0 ∈ <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.refersto0.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.refersto0.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.<name> ≠ undefined) ⇒
(s.surroundingScopeUnit0 ∈ <composite state>) ∨
(s.surroundingScopeUnit0 ∈ <composite state type definition>)

If <state entry point name> is given in a <start>, the <start> must be the <start> of a <composite state>.

Mapping to Abstract Syntax

| s=<start>(*,-entry,exc,trans) => if s.parentAS0 ∈ <procedure body> then mk-Procedure-start-node(Mapping(exc), Mapping(trans))
   else mk-Procedure-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-nodes within a State-transition-graph or Procedure-graph must have different State-name.

∀ de TYPEDEFINITION; ∀ in, in' ∈ Input-node:
(in'in'\in'.parentAS1.s-State-name=in'.parentAS1.s-State-name ∧
in'.parentAS1∈d.stateNodeSet,i=in'.parentAS1∈d.stateNodeSet,)⇒
in.s-Signal-identifier=in'.s-Signal-identifier

The Signal-identifiers 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 b s \in \text{state} \cdot \forall s n e \in \text{name}: (s.nparentAS0 = b.s \cdot \text{state list}) \land b.s \cdot \text{state list} \in s \cdot \text{asterisk state list} \) \Rightarrow

\( (\forall s n e \in \text{name}: (s.nparentAS0 = s.nparentAS0) \Rightarrow (s.n \neq s')) \land

(s.ne bs \cdot \text{surroundingScopeUnit}_0 \cdot \text{stateNameSet}_0)

The \text{name}s in an \text{asterisk state list} must be distinct and must be contained in other \text{state list}s in the enclosing body or in the body of a supertype.

\[ \forall r, r' \in \text{composite state reference}: (rורותferencedDefinition}_0 = r' \cdot \text{referencedDefinition}_0 \Rightarrow

(r.parentAS0 = r'.parentAS0) \lor

(\exists s ne \in \text{state}: \exists s n e \in \text{name}: (s.n \cdot \text{surroundingScopeUnit}_0 \cdot \text{entityName}_0)) //the surrounding scope of r is a composite state.

A \text{composite state reference} to the same composite state must only occur in one of the \text{composite state application}s in the surrounding state machine.

Transformations

\[ < \text{state}>(<s> \cap \text{rest}, \text{exc}, \text{triggers}) > \text{provided rest} \neq \text{empty}

=1=> < \text{state}>(\text{rest}, \text{exc}, \text{triggers}) , < \text{state}>(\text{rest}, \text{exc}, \text{triggers}) >

When the \text{state list} of a \text{state} contains more than one \text{name}, a copy of that \text{state} is created for each such \text{name}. Then the \text{state} is replaced by these copies.

\[ < \text{state}>(<s>, \text{exc1}, \text{triggers1}) > \cap \text{rest} \cap < \text{state}>(<s>, \text{exc2}, \text{triggers2}) >

=13=> \text{rest} \cap < \text{state}>(<s>, \text{if exc1 } \neq \text{undefined then exc1 else exc2 end if}, \text{triggers1} \cap \text{triggers2}) >

When several \text{state}s contain the same \text{name}, these \text{state}s are concatenated into one \text{state} having that \text{name}.

\[ b'=< \text{state}>(<\text{asterisk state list}>(\text{exceptStates}), \text{exc}, \text{triggers})

=13=> < \text{state}>(<s \text{ in sort}(b.\text{surroundingScopeUnit}_0.\text{stateNameSet}_0):s \notin \text{exceptStates}.\text{toSet} >, \text{exc}, \text{triggers})

A \text{state} with an \text{asterisk state list} is transformed to a list of \text{state}s, one for each \text{name} of the body in question, except for those \text{name}s contained in the \text{asterisk state list}.

Mapping to Abstract Syntax

| <state>(<state list item>(name, undefined), exc, triggers) => mk-\text{State-node}(\text{Mapping}(name), \text{Mapping}(exc),

\{ t \in \text{Mapping}(\text{triggers}).\text{toSet}: t \in \text{Save-signalset} \},

\{ t \in \text{Mapping}(\text{triggers}).\text{toSet}: t \in \text{Input-node} \},

\{ t \in \text{Mapping}(\text{triggers}).\text{toSet}: t \in \text{Spontaneous-transition} \},

\{ t \in \text{Mapping}(\text{triggers}).\text{toSet}: t \in \text{Continuous-signal},\text{undefined} \})

| <state>(<typebased composite state>(name, parent), exc, triggers) => mk-\text{State-node}(\text{Mapping}(name), \text{Mapping}(exc),

\{ t \in \text{Mapping}(\text{triggers}).\text{toSet}: t \in \text{Save-signalset} \},

\{ t \in \text{Mapping}(\text{triggers}).\text{toSet}: t \in \text{Input-node} \},

\{ t \in \text{Mapping}(\text{triggers}).\text{toSet}: t \in \text{Spontaneous-transition} \},

\{ t \in \text{Mapping}(\text{triggers}).\text{toSet}: t \in \text{Continuous-signal},\text{Mapping}(parent) \})

Auxiliary Functions

Get the set of \text{name}s of a \text{state}, an agent definition, an agent type definition, a composite state, a composite state type definition or a procedure definition.

\[ \text{stateNameSet}_0(d): \text{state} \cup \text{agent definition} \cup \text{agent type definition} \cup \text{composite state} \cup

\text{composite state type definition} \cup \text{procedure definitions} : <\text{name} > \cdot \text{set} = \text{def} \]
if (de <state>) then
  if d.s.<state list> ∈ <asterisk state list> then
    d.surroundingScopeUnit.stateNameSet \{ 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.stateSet0) ∧ (n ∈ s.stateNameSet0) \}
endif

2.8.3 Input

Abstract Syntax

\[
\text{Input-node} :: \begin{array}{l}
\text{[ PRIORITY ]} \\
\text{Signal-identifier} \\
\text{[ Variable-identifier ]*} \\
\text{[ Provided-expression ]} \\
\text{[ On-exception ]} \\
\text{Transition}
\end{array}
\]

Conditions on Abstract Syntax

\[ \forall in \in \text{Input-node}: \forall sd \in \text{Signal-definition}: \\
sd = \text{getEntityDefinition}(\text{in.s-Signal-identifier}, \text{signal}) \Rightarrow \\
(\forall i_s \in 1..\text{in.s-Variable-identifier.length}: \\
\exists vd \in \text{Variable-definition}: vd = \text{getEntityDefinition}(\text{in.s-Variable-identifier[i], variable}) \land \\
\text{isCompatibleTo}(vd.s-Sort-reference-identifier, sd.s-Sort-reference-identifier[i])) \]

The length of the list of optional Variable-identifiers 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

\[
\text{<input part>} :: \\
\begin{array}{l}
\text{[<virtuality>] <input list> [<on exception>] [<provided expression>] <transition>}
\end{array}
\]

\[
\text{<input list>} = <\text{stimulus}> + | \text{<asterisk input list>}
\]

\[
\text{<stimulus>} :: <\text{signal list item}> [ [ <\text{variable}> ] + | \text{<remote procedure reject> } ]
\]

\[
\text{<remote procedure reject>} :: <\text{raise body}>
\]

\[
\text{<asterisk input list>} :: ()
\]

Conditions on Concrete Syntax

\[ \forall s \in \text{<state>}: | s.asteriskInputListSet0 | \leq 1 \]

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

\[ \forall s \in \text{<state>}: (s.asteriskInputListSet0 = \emptyset) \lor (s.asteriskSaveListSet0 = \emptyset) \]

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

\[ \forall ip \in \text{<input part>}: \forall rpr \in \text{<remote procedure reject>}: \\
\text{isAncestorAS0}(ip, rpr) \Rightarrow rpr.parentAS0.s <\text{signal list item}>.s.<\text{identifier}>.idKind0 = \text{remote procedure} \land \\
(\text{let rpd} = \text{getEntityDefinition}(rpr.parentAS0.s <\text{signal list item}>.s.<\text{identifier}>.remote procedure) \text{ in} \\
rpr.s.<\text{raise body}>.s.<\text{identifier}> \in \text{toSet}(rpd.s.<\text{procedure signature}>.s.<\text{raises}>) \text{ endlet}) \]

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>.
\( \forall ip \in \langle \text{input part} \rangle: \forall \text{sli} \in \langle \text{signal list item} \rangle: \)  
\[ \text{isAncestorAS0}(ip, \text{sli}) \Rightarrow \]
\[ (\text{let } id\text{Kind} = \text{sli}.s.<\text{identifier}>.id\text{Kind}\text{\& } \]
\[ (id\text{Kind} = \text{\& remote variable}) \wedge \]
\[ (id\text{Kind} = \text{\& remote procedure}) \Rightarrow \]
\[ \text{idKind}(\text{sli}.parentAS0.s.<\text{variable}>.seq = \text{empty}) \]
\[ \text{endlet} \]

A \langle \text{signal list item} \rangle must not denote a \langle \text{remote variable identifier} \rangle and if it denotes a \langle \text{remote procedure identifier} \rangle or a \langle \text{signal list identifier} \rangle, the \langle \text{stimulus} \rangle parameters must be omitted.

**Transformations**

\[ \langle \text{stimulus} \rangle(\langle \text{signal list item} \rangle,(\text{signallist}, id), todo) > \]
\[ = 8 => \langle \text{stimulus} \rangle(\langle \text{signal list item} \rangle, sig, todo) | \text{sig in refersto}, s.<\text{signal list item}>.seq \]

A \langle \text{stimulus} \rangle whose \langle \text{signal list item} \rangle is a \langle \text{signal list identifier} \rangle is derived syntax for a list of \langle \text{stimulus} \rangles without parameters and is inserted in the enclosing \langle input list \rangle or \langle priority input list \rangle. In this list, there is a one to one correspondence between the \langle stimulus \rangles and the members of the signal list.

\[ \langle \text{input part} \rangle(virt, <\text{stim}>, \text{rest}, exc, cond, trans) > \text{provided rest} \neq \text{empty} \]
\[ = 1 => \langle \text{input part} \rangle(virt, <\text{stim}>, \text{rest}, exc, cond, trans), \langle \text{input part} \rangle(virt, rest, exc, cond, trans) > \]

When the \langle \text{stimulus} \rangles list of an \langle \text{input part} \rangle contains more than one \langle \text{stimulus} \rangle, a copy of the \langle \text{input part} \rangle is created for each such \langle \text{stimulus} \rangle. Then the \langle \text{input part} \rangle is replaced by these copies.

\[ \langle \text{input part} \rangle(virt, <\text{stimulus}>(\langle \text{item}, \text{vars} \rangle), exc, cond, <\text{transition gen action statement}>(\langle \text{actions}, \text{term} \rangle)) \]
\[ \text{provided } \text{vars} \notin \langle \text{remote procedure reject} \rangle \wedge \]
\[ \{ \text{v in } \text{vars.toSet}: \text{v in } (\langle \text{indexed variable} \rangle \cup \langle \text{field variable} \rangle) \} \neq \emptyset \]
\[ = 8 => \]
\[ \text{let newName =} \]
\[ \langle \text{if } \text{v in } (\langle \text{indexed variable} \rangle \cup \langle \text{field variable} \rangle) \text{ then newName(v) else } \text{v endif} \rangle | \text{v in } \text{vars} \rangle \text{ in } \]
\[ \text{let newtrans =} \langle \text{transition gen action statement}>(\]
\[ \langle \text{if } \text{action statement}(\langle \text{undefined}, \text{task} \rangle(\langle \text{assignment}>(\langle \text{v, newName(v)\rangle, undefined}) \]
\[ | \text{v in } \text{vars}: (\text{v in } (\langle \text{indexed variable} \rangle \cup \langle \text{field variable} \rangle)) \rangle \]
\[ \langle \text{actions, term} \rangle \text{ in } \]
\[ \langle \text{input part} \rangle(virt, <\text{stimulus}>(\langle \text{item, newvars} \rangle), exc, cond, newtrans \rangle \]
\[ \text{endlet} \]

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

The following statement is handled by the dynamic semantics.

An \langle asterisk input list \rangle is transformed to a list of \langle input part \rangles, one for each member of the complete valid input signal set of the enclosing \langle agent definition \rangle, except for \langle signal identifier \rangles of implicit input signals introduced by the concepts in 10.5, 10.6, 11.4, 11.5, and 11.6 and for \langle signal identifier \rangles contained in the other \langle input list \rangles and \langle save list \rangles of the \langle state \rangle.

**Mapping to Abstract Syntax**

\[ \langle \text{input part} \rangle(\langle *, <\text{stimulus}>(\langle \text{item, vars} \rangle), exc, cond, trans \rangle) \]
\[ = \text{mk-Input-node}(\text{undefined}, \text{Mapping(item)}, \]
\[ \text{Mapping(vars), Mapping(cond), Mapping(exc), Mapping(trans)}) \]

**Auxiliary Functions**

Get the \langle asterisk input list \rangle for a \langle state \rangle.

\[ \text{asteriskInputListSet}(s: <\text{state}) >: \langle \text{asterisk input list} \rangle => \text{def} \]
\[ \{ \text{ai in } \langle \text{asterisk input list} \rangle: \text{isAncestorAS0}(s, \text{ai}) \} \]
2.8.4 Priority Input

Concrete Syntax

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

```plaintext
<priority input list> = <stimulus>+
```

Mapping to Abstract Syntax

```plaintext
| <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

```plaintext
Continuous-signal :: [ On-exception ]
    Continous-expression
[ Priority-name ]
    Transition

Continuous-expression = Boolean-expression
Priority-name = NAT
```

Concrete Syntax

```plaintext
<continuous signal> ::= 
    [<virtuality>] <continuous expression> [<priority name>] [<on exception>] <transition>
```

```plaintext
<continuous expression> = <Boolean-expression>
<priority name> = <Natural-name>
```

Mapping to Abstract Syntax

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

2.8.6 Enabling Condition

Abstract Syntax

```plaintext
Provided-expression = Boolean-expression
Boolean-expression = Expression
```

Concrete Syntax

```plaintext
<provided expression>:: <Boolean-expression>
```

Transformations

```plaintext
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().getEntities
    => entities ∩
    <procedure definition>(empty,
        <procedure heading>(<procedure preamble>(undefined, undefined), empty, nn, empty, undefined, undefined, empty, <result>("Boolean"), empty),
```

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

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

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

∀s ∈ <state>: |s.asteriskSaveListSet| ≤ 1

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

∀s ∈ <state>: (s.asteriskInputListSet = ∅) ∨ (s.asteriskSaveListSet₀ = ∅)

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>.

```
asteriskSaveListSet₀(s: <state>): <asterisk save list>-set = def
    {asl ∈ <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

\[
\text{Spontaneous-transition} :: [\text{On-exception}] [\text{Provided-expression}] \text{Transition}
\]

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

\[
\text{Free-action} :: \text{Connector-name} \text{Transition}
\]

Concrete Syntax

<label> :: <connector><name>

<free action> :: <transition>

Conditions on Concrete Syntax

\[\forall b \in \text{<composite state body>} \cup \text{<operation body>} \cup \text{<procedure body>} \cup \text{<agent body>}: \forall l, l' \in \text{<label>}: (\text{isAncestorAS0}(b,l) \land \text{isAncestorAS0}(b,l') \land (l \neq l') \Rightarrow (l.s.<name> \neq l'.s.<name>))\]

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

\[\forall fa \in \text{<free action>}, \forall l \in \text{<label>}: \\
((l = fa.s.<transition>.s.<terminator statement>.s.<label> ) \lor \\
(l = fa.s.<transition>.s.<action statement>-\text{seq}.head.s.<label>)) \Rightarrow \\
(l.s.<name> = fa.s.<name>)\]

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.

\[
\text{getLabel}(t: \text{<transition>}): \text{<label>} = \text{def} \\
\begin{align*}
\text{If } & t.s.<\text{action statement}> = \text{empty} \text{ then } t.s.<\text{terminator statement}>.s.<\text{label}> \\
\text{else } & t.s.<\text{action statement}>.\text{head}.s.<\text{label}> \\
\text{endif}
\end{align*}
\]
Transformations

\(<a, s> = \langle\text{action statement}\rangle(l, *) > \cap r \text{ provided } l \neq \text{undefined} = 5 = > <a >

\text{and}

\(a.\text{parentAS0} \Rightarrow \langle\text{transition gen action statement}\rangle(a.\text{parentAS0}.s> \langle\text{action statement}\rangle,
\langle\text{terminator statement}\rangle(\text{undefined}, \langle\text{join}\rangle(l.s<\text{name}))))

\text{and}

\text{let } p = \text{parentAS0ofKind}(a, \langle\text{free action}\rangle \cup \langle\text{state}\rangle) \text{ in}
\langle p > = > p, \langle\text{free action}\rangle(\langle s > \cap r >)

If a \(<\text{label}\>) is not the first label of a \(<\text{transition string}\>\), the \(<\text{transition string}\>\) is split into two parts. All \(<\text{action statement}\>\>s preceding the \(<\text{label}\>\) are preserved in the original transition, which is terminated with a \(<\text{join}\>\) to the \(<\text{label}\>\). All action statements following \(<\text{label}\>\) are copied to a new \(<\text{free action}\>\), which starts with the \(<\text{label}\>\).

Mapping to Abstract Syntax

\[ | \langle\text{free action}\rangle(\text{trans}) => \text{mk-Free-action}(\text{Mapping(getLabel(\text{trans}))}, \text{Mapping(\text{\text{trans}}))) \]

### 2.8.11 State Machine and Composite State

#### Abstract Syntax

\[
\begin{align*}
\text{Composite-state-formal-parameter} & = \text{Agent-formal-parameter} \\
\text{State-entry-point-definition} & = \text{Name} \\
\text{State-exit-point-definition} & = \text{Name} \\
\text{Exit-procedure-definition} & = \text{Procedure-definition} \\
\text{Entry-procedure-definition} & = \text{Procedure-definition} \\
\text{Named-start-node} & :: \text{State-entry-point-name} \\
& \text{[ On-exception ]} \\
& \text{Transition}
\end{align*}
\]

#### Conditions on Abstract Syntax

\(\forall d \in \text{Entry-procedure-definition}: (d.s-\text{Procedure-name} = \text{“entry”}) \land (d.\text{formalParameterList}_1 = \text{empty}) \land (d.\text{stateNodeSet}_1 = \emptyset)\)

\(\forall d \in \text{Exit-procedure-definition}: (d.s-\text{Procedure-name} = \text{“exit”}) \land (d.\text{formalParameterList}_1 = \text{empty}) \land (d.\text{stateNodeSet}_1 = \emptyset)\)

\(\text{Entry-procedure-definition}\) represents a procedure with the name entry. \(\text{Exit-procedure-definition}\) represents a procedure with the name exit. These procedures may not have parameters, and may only contain a single transition.

#### Concrete Syntax

\[
\langle\text{composite state}\rangle ::
\text{<package use clause>* <composite state heading> <composite state structure>}
\]

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

\[
\text{<composite state heading>} ::
\text{<qualifier> <composite state<name> <formal agent parameter>*}
\]

\[
\text{<entity in composite state>} =
\text{<variable definition>}
\text{| <data definition>}
\text{| <select definition>}
\]
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

∀csg ∈ <composite state body>:
∃!s ∈ <start>: (s.parentAS0 = csg ∧ (s.s-<name> = undefined))

Exactly one of the <start>s shall be unlabelled.

∀csd ∈ <composite state> ∪ <composite state type definition>:
(∀pn ∈ <name>: pn ∈ csd.usedEntryNameSet0 ⇒ pn ∈ csd.definedEntryNameSet0)
∧ (∀pn ∈ <name>: pn ∈ csd.usedExitNameSet0 ⇒ pn ∈ csd.definedExitNameSet0)

Each additional labelled entry and exit point must be defined by a corresponding <state connection points>.

∀csb ∈ <composite state body>: ∀s ∈ <state>: (s ∈ csb.surroundingScopeUnit0.stateSet0 ∧
(s.s-<state list> ∉ <asterisk state list>)) ⇒ csb.surroundingScopeUnit0.startSet0 ≠ ∅

If a <composite state body> contains at least one <state> different from asterisk state, a <start> must be present.

∀cs ∈ <composite state> : ∀vd ∈ <variable definition>:
(vd.surroundingScopeUnit0 ∈ cs) ∧ (cs.surroundingScopeUnit0 ∈ <procedure definitions>) ⇒
(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 ∩ <state>(<asterisk state list>(undefined), exc, triggers, undefined) ∩ items2)
provided < i in (items1 ∩ items2): (i ∈ <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 ∩ <state>(< nn >, exc, triggers, undefined) ∩ items2)
endlet)
```

```
<composite state body>(exc, empty,
  il ∩ <state>(<asterisk state list>(undefined), exc, triggers, undefined) ∩ i2)
provided < i in (items1 ∩ items2): (i ∈ <state>) = empty
=8=>
```
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 , conn , entities) , <composite state body>(exc , starts , items)) => mk-Compatible-state-graph

Auxiliary Functions

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

\[
\text{usedEntryNameSet}(csd: <\text{composite state}>|<\text{composite state type definition}>):<\text{name}-\text{set}=\{n \in <\text{name}>: n.\text{parentAS0} \in \text{csd.startSet0}\}
\]

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

\[
\text{usedExitNameSet}(csd: <\text{composite state}>|<\text{composite state type definition}>):<\text{name}-\text{set}=\{n \in <\text{name}>:\exists s \in \text{csd.stateSet0}: \text{isAncestorAS0}(s,n)\land (n.\text{parentAS0}.\text{parentAS0} \in <\text{return}>)}\]

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

\[
\text{definedEntryNameSet}(csd: <\text{composite state}>|<\text{composite state type definition}>):<\text{name}-\text{set}=\{n \in <\text{name}>: (n.\text{parentAS0} \in <\text{state entry points})\land (n.\text{parentAS0}.\text{parentAS0} = \text{csd.s-<composite state structure>})\}
\]

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

\[
\text{definedExitNameSet}(csd: <\text{composite state}>|<\text{composite state type definition}>):<\text{name}-\text{set}=\{n \in <\text{name}>: (n.\text{parentAS0} \in <\text{state exit points})\land (n.\text{parentAS0}.\text{parentAS0} = \text{csd.s-<composite state structure>})\}
\]

2.8.11.2 State Aggregation

Abstract Syntax

\[
\text{State-aggregation-node} :: \text{State-partition*}
\]

\[
\text{State-partition} :: \text{Name} \cdot \text{Composite-state-type-identifier} \cdot \text{Connection-definition-set}
\]

\[
\text{Connection-definition} = \text{Entry-connection-definition} \lor \text{Exit-connection-definition}
\]

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

∀ pn ∈ State-entry-point-name: (pn.parentAS1 ∈ Outer-entry-point) ⇒
(pn ∈ pn.surroundingScopeUnit0.entryPointSet1)∧
(pn.surroundingScopeUnit0.s-implicit ∈ State-aggregation-node)

The State-entry-point-name in the Outer-entry-point must denote a State-entry-point-definition of the Composite-state-type-definition where the State-aggregation-node occurs.

∀ pn ∈ State-entry-point-name: (pn.parentAS1 ∈ Inner-entry-point) ⇒
(pn.parentAS1.ofKind(pn, State-partition).baseType1.entryPointSet1)

The State-entry-point-name of the Inner-entry-point must denote a State-entry-point-definition of the composite state in the State-partition.

∀ pn ∈ State-exit-point-name: (pn.parentAS1 ∈ Outer-exit-point) ⇒
(pn ∈ pn.surroundingScopeUnit0.exitPointSet1)∧
(pn.surroundingScopeUnit0.s-implicit ∈ State-aggregation-node)

∀ pn ∈ State-exit-point-name: (pn.parentAS1 ∈ Inner-exit-point) ⇒
(pn.parentAS1.ofKind(pn, State-partition).baseType1.exitPointSet1)

Likewise, the Outer-exit-points must denote exit points in the inner and outer composite state, respectively.

∀ td ∈ Composite-state-type-definition: (td.s-implicit ∈ State-aggregation-node)⇒
∃! cd ∈ Connection-definition: (cd.surroundingScopeUnit0=td)∧
(let pointSet = {pn ∈ State-entry-point-definition→State-exit-point-definition:
(pn ∈ td.entryPointSet1∪td.exitPointSet1)∨
(∃sp ∈ State-partition: (sp.surroundingScopeUnit0= td)∧
 (pn ∈ sp.baseType1.entryPointSet1∪sp.baseType1.exitPointSet1))) in
let pointSet'={pn ∈ State-entry-point-definition→State-exit-point-definition: isAncestorAS1(cd,pn)} in
pointSet ⊆ pointSet'
endlet)

All entry and exit points of the both the container state and the state partitions must appear in exactly one Connection-definition.

∀ sp, sp' ∈ State-partition: (sp≠sp')∧ (sp.parentAS1= sp'. parentAS1)⇒
sp.inputSignalSet1∩ sp'. inputSignalSet1=∅

The input signal sets of the State-partitions 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 = ⋂<n > ⋂*),
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.identifier0) > = empty
=⇒>
<composite state structure>(inset, gates, p, entities,
<state partition connection gen entry>=(<entry point><undefined, n),
<entry point>=(s.identifier0, default)) ⊂ body)

<composite state structure>(inset, gates, p = ⋂<state exit points>(* ⋂<n > ⋂*),
entities, body = ⋂<n > ⋂*),
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.identifier0) > = empty
=⇒>
<composite state structure>(inset, gates, p, entities,
<state partition connection gen exit>=(<exit point><undefined, n),
<exit point>=(s.identifier0, 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>(inset, gates, p = ⋂<state entry points>(* ⋂<n > ⋂*),
entities, body = ⋂<n > ⋂*),
uses, <composite state heading>(*, n, params), struct)>
=⇒<
<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,(sp: TYPEDEFINITION ∪ State-partition ∪ State-node): SIGNAL₁ = def

If sp State-node then
sp.S-Input-node-set \{in.S-Signal-identifier: in ∈ sp.S-Input-node-set\} \cup
{getEntityDefinition(cn.S-Procedure-identifier, procedure).inputSignalSet;
 cn ∈ {cn ∈ Call-node: isAncestorASI(sp, cn)}}
else // sp TYPEDEFINITION ∪ State-partition
{sn.InputSignalSet1; sn ∈ sp.stateNodeSet1}
endf

Get the base type of a definition.

baseType₁(as: Agent-definition ∪ State-machine-definition ∪ State-partition ∪ State-node):

Agent-definition ∪ Composite-state-type-definition = def
case as of
| Agent-definition => getEntityDefinition.(as.s-Agent-type-identifier agent type)
| State-machine-definition ∪ State-partition =>
  getEntityDefinition.(as.s-Composite-state-type-identifier, state type)
| State-node =>
  if as.s-Composite-state-type-identifier = undefined then undefined
  else getEntityDefinition.(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.

case d of
| TYPEDEFINITION => d.localStateNodeSet1 ∪ d.inheritedStateNodeSet1
| State-node =>
  if d.s-Composite-state-type-identifier ≠ undefined then d.baseType1.stateNodeSet1 ∪ {d}
  else {d}
| State-partition => d.baseType1.stateNodeSet1
otherwise =⇒ Ø
endcase

Get the set of state nodes defined locally in a type definition.

localStateNodeSet1(d: TYPEDEFINITION): State-node-set = def
case d of
| Agent-type-definition =>
  if d.s-State-machine-definition ≠ undefined then
d.s-State-machine-definition.baseType1.stateNodeSet1
  else Ø
| Procedure-definition =>
  {sn.stateNodeSet1: sn ∈ d.s-Procedure-graph.s-State-node-set}
| Composite-state-type-definition =>
  if d.s-implicit Composite-state-graph then
  {sn.stateNodeSet1: sn ∈ d.s-implicit.s-State-transition-graph.s-State-node-set}
  else // d.s-implicit State-aggregation-node
  {sp.stateNodeSet1: sp ∈ d.s-implicit.s-State-partition-seq.toSet}
otherwise =⇒ Ø
endcase

Get the set of state nodes defined in a super type.

inheritedStateNodeSet1(d: TYPEDEFINITION): State-node-set = def
case d of
| Agent-type-definition =>
  if d.s-Agent-type-identifier ≠ undefined then
gEntityDefinition.(d.s-Agent-type-identifier, agent type).stateNodeSet1
  else Ø
| Procedure-definition =>
  if d.s-Procedure-identifier ≠ undefined then
gEntityDefinition.(d.s-Procedure-identifier, procedure).stateNodeSet1
  else Ø
| Composite-state-type-definition =>
  if d.s-Composite-state-type-identifier ≠ undefined then
gEntityDefinition.(d.s-Composite-state-type-identifier, state type).stateNodeSet1
  else Ø
otherwise =⇒ Ø
endcase

Get the set of the state entry points of a Composite-state-type-definition.
Get the set of the state exit points of a Composite-state-type-definition.

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

\(<\text{state connection points}> = <\text{state entry points}> | <\text{state exit points}>\\n<\text{state entry points}> :: <\text{state entry point}<\text{name}>+\\n<\text{state exit points}> :: <\text{state exit point}<\text{name}>+

Mapping to Abstract Syntax

| <\text{state entry points}>(x) => Mapping(x)\\n| <\text{state exit points}>(x) => Mapping(x)

2.8.11.4 Connect

Abstract Syntax

Connect-node :: [ \text{State-exit-point-name} ] [ \text{On-exception} ] Transition

Concrete Syntax

<\text{connect part}> ::\\n[\text{<virtuality}>] <\text{connect list}> [\text{<on exception}>] <\text{transition}>

<\text{connect list}> = <\text{state exit point}<\text{name}>* | \text{asterisk connect list}\\n<\text{asterisk connect list}> :: ()

Conditions on Concrete Syntax

\(\forall cl \in <\text{connect list}>: \forall pn \in <\text{name}>:\) isAncestorAS0(cl, pn) \Rightarrow\\n(\exists scp \in <\text{state connection points}>: \exists sep \in <\text{state exit points}>:\)

isAncestorAS0(scp, sep) \land isAncestorAS0(scp.parentAS0, pn) \land (pn = sep)

The <\text{connect list}> must only refer to visible <\text{state exit point}>s.

Transformations

< <\text{connect part}>(\text{virt}, <n> ^* \text{rest}, \text{exc}, \text{trans}) > \text{provided rest \neq empty}\\n=1=> < <\text{connect part}>(\text{virt}, <n>, \text{exc}, \text{trans}), <\text{connect part}>(\text{virt}, \text{rest}, \text{exc}, \text{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.

\[
c = \langle \text{connect part}(\text{virt}, \langle \text{asterisk}, \text{exc}, \text{trans} \rangle) \rangle
\]

\[
= \text{\texttt{let}} \ \text{parentType} = \text{c.parentAS0.s-\text{state list}.head.s-\text{type expression}.refersto0 in}
\]

\[
\text{let allExits = BigSeq(\langle \text{ex.s-\text{name}} \rangle | \text{ex in exits} \rangle | \text{exits in parentType.s-\text{composite state structure}.s-\text{state connection points}-seq: (exits \in \text{\langle state exit points\rangle}) >) in}
\]

\[
\langle \text{connect part}(\text{virt}, \text{allExits}, \text{exc}, \text{trans} \rangle \rangle
\]

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

\[
| \langle \text{connect part}\rangle(\langle, \text{name}, \text{exc}, \text{trans} \rangle) => \text{mk-Connect-node}(\text{\texttt{Mapping}}(\text{\texttt{name}}), \text{\texttt{Mapping}}(\text{\texttt{exc}}), \text{\texttt{Mapping}}(\text{\texttt{trans}}))
\]

### 2.8.12 Transition

#### 2.8.12.1 Transition Body

**Abstract Syntax**

\[
\begin{align*}
\text{Transition} & \ :: \ Graph-node* \ {\Terminator | Decision-node} \\
\text{Graph-node} & \ :: \ \{ \ \text{Task-node} \\
& | \ \text{Output-node} \\
& | \ \text{Create-request-node} \\
& | \ \text{Call-node} \\
& | \ \text{Compound-node} \\
& | \ \text{Set-node} \\
& | \ \text{Reset-node} \\
& \ \} \ [ \text{On-exception} ] \\
\text{Terminator} & \ :: \ \{ \ \text{Nextstate-node} \\
& | \ \text{Stop-node} \\
& | \ \text{Return-node} \\
& | \ \text{Join-node} \\
& | \ \text{Continue-node} \\
& | \ \text{Break-node} \\
& | \ \text{Raise-node} \\
& \ \} \ [ \text{On-exception} ]
\end{align*}
\]

**Concrete Syntax**

\[
<\text{transition}> = <\text{transition gen action statement}> | <\text{terminator statement}>
\]

\[
<\text{transition gen action statement}> :: <\text{action statement}>+ [ <\text{terminator statement}> ]
\]

\[
<\text{action statement}> :: [ <\text{label}> ] <\text{action}>
\]

\[
<\text{action}> =
\]

\[
| <\text{task}>
| <\text{output}>
| <\text{create request}>
| <\text{decision}>
| <\text{set}>
\]
<terminator statement> :: [label] <terminator>

<terminator> = <nextstate> | <join> | <stop> | <return> | <raise>

Conditions on Concrete Syntax

∀ t ∈ <transition>: (t ∉ <terminator statement>) ∧ (t.s = undefined) ∧
(t.parentAS0.parentAS0.parentAS0 ∉ <decision> ∪ <transition option>) ⇒
(let asl = t.s<action statement>-seq in
   (asl.last.s<action> ∈ <decision> ∪ <transition option>) ∧
   isTransitionTerminating(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<transition gen action statement>(*,*) provided t.parentAS0 ∉ <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 ∉ <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.

\[
\text{isTransitionTerminating}_0(dt: \text{<decision>∪<transition option>}): \text{BOOLEAN} = \exists t \in \text{<transition>}:
\]
\[
((t.e <\text{terminator statement}>) \lor ((t.e <\text{terminator statement}) \neq \text{undefined}) \lor
\]
\[
((t.e <\text{terminator statement}) \land (t.s <\text{terminator statement}) = \text{undefined}) \land
\]
\[
\text{isTransitionTerminating}_0(t.s <\text{action statement} > - \text{seq}.last.<\text{action}>)
\]

2.8.12.2 Nextstate

Abstract Syntax

\[
\begin{align*}
\text{Nextstate-node} &= \text{Name-nextstate} \mid \text{Dash-nextstate} \\
\text{Dash-nextstate} :: [\text{HISTORY}] \\
\text{Name-nextstate} :: \text{State-name} [\text{Nextstate-parameters}] \\
\text{Nextstate-parameters} :: [\text{Expression}]^* [\text{State-entry-point-name}]
\end{align*}
\]

Conditions on Abstract Syntax

\[
\forall nn \in \text{Nextstate-node}: nn.s\cdot\text{Nextstate-parameters} \neq \text{undefined} \Rightarrow
\]
\[
(nn.s\cdot\text{State-name} = sn.s\cdot\text{State-name}) \land
\]
\[
(\text{parentAS1ofKind}(nn, \text{State-transition-graph∪Procedure-graph}) =
\]
\[
\text{parentAS1ofKind}(sn, \text{State-transition-graph∪Procedure-graph})) \land
\]
\[
(nn.s\cdot\text{Nextstate-parameters} \neq \text{undefined} \Rightarrow sn.s\cdot\text{Composite-state-type-identifier} \neq \text{undefined})
\]

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

\[
<\text{nextstate}> = <\text{nextstate body}>
\]
\[
<\text{nextstate body}> = <\text{nextstate body gen name}> \mid <\text{dash nextstate}> \mid <\text{history dash nextstate}>
\]
\[
<\text{nextstate body gen name}> ::
\]
\[
<\text{state}<\text{name}>[<\text{actual parameter}>]^*[<\text{state entry point}<\text{name}>]
\]
\[
<\text{dash nextstate}> :: ()
\]
\[
<\text{history dash nextstate}> :: ()
\]

Conditions on Concrete Syntax

\[
\forall \text{se } <\text{state}>: \forall \ell <\text{transition}>:
\]
\[
((t.e <\text{parentAS0} <\text{input part} > \cup <\text{priority input} > \cup <\text{spontaneous transition} > \cup <\text{continuous signal}>) \land
\]
\[
((t.e <\text{parentAS0} <\text{parentAS0} = s)) \land
\]
\[
(\exists hdn <\text{history dash nextstate}> : \text{isReachableFrom}_0(hdn, t) \Rightarrow
\]
\[
(s.e <\text{surroundingScopeUnit}_0 <\text{composite state} > \cup <\text{composite state type definition}>)
\]

If a transition is terminated by a <history dash nextstate>, the <state> must be a <composite state>.

\[
\forall \ell <\text{transition}> : \forall ne <\text{dash nextstate}> : (t.e <\text{parentAS0} <\text{start}> ) \Rightarrow \neg \text{isReachableFrom}_0(n, t)
\]

The <transition> contained in a <start> must not lead, directly or indirectly, to a <dash nextstate>.

\[
\forall \ell <\text{transition}> : \forall ne <\text{history dash nextstate}> : (t.e <\text{parentAS0} <\text{start}> \cup <\text{handle} > ) \Rightarrow
\]
\[
\neg \text{isReachableFrom}_0(n, t)
\]

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 ∈ oereachableExceptionHandlerSet0) ⇒
(∀dn ∈ <dash nextstate>: ∀t ∈ <transition>: t.parentAS0 in eh.s <handle>-seq⇒
¬isReachableFrom0(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, priority inputs, continuous signals, enabling conditions, implicit tasks for imperative actions and remote variables 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 exceptionHandlerNameSet0 gets the set of <exception handler name>s appeared in the given definition.

```plaintext
excepcionHandlerNameSet0( ed: <exception handler∪<agent definition>∪<agent type definition>∪
<composite state>∪<composite state type definition>∪<procedure definitions>): <name>-set =
if ed ∈ exception handler then
  if ed.s. <exception handler list> ∈ <asterisk exception handler list> then
    ed.s. surroundingScopeUnit0, exceptionHandlerNameSet0
  else ed.s. <exception handler list>. toSet
  endif
else
  {n ∈ <name>: ∃eh ∈ <exception handler>
  n ∈ eh. exceptionHandlerNameSet0, eh ∈ ed. exceptionHandlerSet0}
endif
```

The function reachableExceptionHandlerSet0 gets the set of <exception handler>s reachable from an <on exception>.
reachableExceptionHandlerSet(oe: <on exception>): <exception handler>-set = def 
{eh∈ <exception handler>: 
( eh = oe.s.<name>.associatedExceptionHandler ) ∨ 
( ∃ oe'∈ <on exception>: 
  isAncestorAS0(oe.s.<name>.associatedExceptionHandler, oe') ∧ 
  ( eh ∈ oe'.reachableExceptionHandlerSet() ) )
}

The function isReachableFrom determines if a <nextstate> is reachable from a <transition>.

isReachableFrom0(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>) ∧ 
    isReachableFrom0(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

\[ \text{Return-node} \quad = \quad \text{Action-return-node} \]
\[ \quad | \quad \text{Value-return-node} \]
\[ \quad | \quad \text{Named-return-node} \]
\[ \text{Action-return-node} \quad :: \quad () \]
\[ \text{Value-return-node} \quad :: \quad \text{Expression} \]
\[ \text{Named-return-node} \quad :: \quad \text{State-exit-point-name} \]

Conditions on Abstract Syntax

\[ \forall \text{rn} \in \text{Return-node} \exists \text{pg} \in \text{Procedure-graph} : \text{isAncestorAS1} (\text{pg}, \text{rn}) \]

A Return-node must be contained in a Procedure-graph.

\[ \forall \text{rn} \in \text{Action-return-node} \exists \text{d} \in \text{Procedure-definition} : \]
\[ \text{isAncestorAS1} (\text{d}.\text{s-Procedure-graph}, \text{rn}) \land \text{d}.\text{s-Result} = \text{undefined} \]

An Action-return-node must only be contained in the Procedure-graph of a Procedure-definition without Result.

\[ \forall \text{rn} \in \text{Value-return-node} \exists \text{d} \in \text{Procedure-definition} : \]
\[ \text{isAncestorAS1} (\text{d}.\text{s-Procedure-graph}, \text{rn}) \land \text{d}.\text{s-Result} \neq \text{undefined} \]

A Value-return-node must only be contained in the Procedure-graph of a Procedure-definition containing Result.

\[ \forall \text{rn} \in \text{Named-return-node} \exists \text{sg} \in \text{Composite-state-graph} : \text{isAncestorAS1} (\text{sg}, \text{rn}) \]

A Named-return-node must only be contained in a Composite-state-graph.

Concrete Syntax

\[ <\text{return}> :: <\text{return body}> [ <\text{on exception}> ] \]

\[ <\text{return body}> :: [ <\text{expression}> | <\text{state exit point}<\text{name}> ] \]

Mapping to Abstract Syntax

\[ | <\text{return}> (x, \text{exc}) \Rightarrow \begin{cases} \text{if } x = \text{undefined} \text{ then } \text{mk-Terminator}(\text{mk-Action-return-node}(), \text{Mapping}(\text{exc})) \\ \text{elseif } x \in \text{<name>} \text{ then } \text{mk-Terminator}(\text{mk-Named-return-node}(\text{Mapping}(x)), \text{Mapping}(\text{exc})) \\ \text{else } \text{mk-Terminator}(\text{mk-Value-return-node}(\text{Mapping}(x)), \text{Mapping}(\text{exc})) \end{cases} \]

endif

2.8.12.6 Raise

Abstract Syntax

\[ \text{Raise-node} \quad :: \quad \text{Exception-identifier} [ \text{Expression} ]^* \]

Conditions on Abstract Syntax

\[ \forall \text{rn} \in \text{Raise-node} \exists \text{d} \in \text{Exception-definition} : \]
\[ ( \text{d}.\text{getEntityDefinition}(\text{rn}.\text{s-Exception-identifier}.\text{exception}) ) \Rightarrow \]
\[ (\text{rn}.\text{s-Expression-seq}.\text{length} = \text{d}.\text{s-Sort-reference-identifier-seq}.\text{length}) \land \]
\[ (\forall i \in 1..\text{rn}.\text{s-Expression-seq}.\text{length} : \]
\[ \text{rn}.\text{s-Expression}[i] = \text{undefined} \lor \]
\[ \text{isCompatibleTo}_1 (\text{rn}.\text{s-Expression}[i].\text{staticSort}_1, \text{d}.\text{s-Sort-reference-identifier}[i]))) \]

In a Raise-node, the length of the list of optional Expressions 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

\[
\text{\texttt{raise}} :: \text{\texttt{raise body}} [\text{on exception}]
\]

\[
\text{\texttt{raise body}} :: \text{\texttt{exception<identifier> [actual parameter]}*}
\]

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

\[
| \text{\texttt{raise}}(r, exc) => \text{\texttt{mk-Terminator}}(\text{\texttt{Mapping}}(r), \text{\texttt{Mapping}}(exc)) \\
| \text{\texttt{raise body}}(id, param) => \text{\texttt{mk-Raise-node}}(\text{\texttt{Mapping}}(id), \text{\texttt{Mapping}}(param))
\]

2.8.13 Action

2.8.13.1 Task

Abstract Syntax

\[
\text{\text{\texttt{Task-node}}} = \text{\text{\texttt{Assignment}}} \\
| \text{\text{\texttt{Assignment-attempt}}} \\
| \text{\text{\texttt{Informal-text}}}
\]

Concrete Syntax

\[
\text{\texttt{task}} :: \{ \text{\texttt{assignment}} | \text{\texttt{informal text}} | \text{\texttt{compound statement}} \} [\text{on exception}]
\]

Transformations

\[
< \text{\texttt{task}}(\text{\texttt{compound statement}})(\text{\texttt{statement list}}(*, empty)) > \\
=1=> empty
\]

If the \text{\texttt{statement list}} in the \text{\texttt{compound statement}} of \text{\texttt{textual task body}} is empty, then the \text{\texttt{task}} is removed. If the \text{\texttt{statement list}} in a \text{\texttt{graphical task body}} is empty, the \text{\texttt{task area}} is removed. Any syntactic item leading to the \text{\texttt{task}} or \text{\texttt{task area}} shall then lead directly to the item following the \text{\texttt{task}} or \text{\texttt{task area}}, respectively.

The following statement is handled by the Mapping of \text{\texttt{compound statement}}.

A \text{\texttt{task}} containing a \text{\texttt{compound statement}} is transformed as shown in 11.14.1. The result of this transformation is inserted in place of \text{\texttt{task}}.

NOTE – The transform of a \text{\texttt{task}} or \text{\texttt{task area}} containing a \text{\texttt{statement list}} is not necessarily mapped onto a Task-node in the Abstract Grammar.

Mapping to Abstract Syntax

\[
| \text{\texttt{task}}(t, exc) => \text{\texttt{mk-Graph-node}}(\text{\texttt{Mapping}}(t), \text{\texttt{Mapping}}(exc))
\]

2.8.13.2 Create

Abstract Syntax

\[
\text{\texttt{Create-request-node}} :: \{ \text{\texttt{Agent-identifier}} | \textbf{THIS} \} \\
[ \text{\texttt{Expression}]}^*
\]

Conditions on Abstract Syntax

\[
\forall n \in \text{\texttt{Create-request-node}}: \forall d \in \text{\texttt{Agent-definition}}: \text{\texttt{(}}d=\text{\texttt{getEntityDefinition}}(n.*\text{\texttt{-Agent-identifier,agent}})\text{\texttt{)}}\Rightarrow
\]

ITU-T Z.100/Annex F2 (11/2000) 121
The length of the list of optional Expressions must be the same as the number of Agent-formal-parameters 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

∀ cre ∈ <create body>: (cr.s-implicit = this) ⇒
(cr.surroundingScopeUnit ∈ <agent type definition>) ∧
(cr.surroundingScopeUnit.surroundingScopeUnit ∈ <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.

\[
c = \text{create request}(\text{create body}(id, params), exc)
\]

provided id.refersto ∈ <agent type definition> ∧ c.possibleInstances ≠ empty
=⇒
(let inst = c.possibleInstances in
  if inst.length = 1 then create request(create body(inst.head.identifier, params), exc)
  else decision(any, exc, decision body(<
    <answer part>(undefined,
      <transition gen action statement><action statement>(undefined,
        create request(create body(elem.identifier, 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.

\[
\text{let } nn = \text{newName in}
\]
\[
c = \text{create request}(\text{create body}(id, params), exc)
\]

provided id.refersto ∈ <agent type definition> ∧ c.possibleInstances = empty
=⇒
create request(create body(create identifier(underscore, nn), params), exc)

and
entities = parentAS0ofKind(c, <agent definition> ∪ <agent type definition>).getEntities
⇒ entities ⊇
if id.refersto ∈ <system type definition>
then <textual typebased system definition>(

\[
(∃ t ∈ Agent-type-definition:
  (t = \text{getEntityDefinition}(d.s-Agent-type-identifier,agent type)) ∧
  \text{isActualAndFormalParameterMatched}(n.s-Expression-seq, t.formalParameterSortList,1))
\]
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.

\[
\text{possibleInstances}(c; \text{ <create request> }): \text{ <agent definition>* } = \text{def} \\
\quad \langle e \text{ in parentAS0ofKind}(c; \text{ <agent definition> } \cup \text{ <agent type definition> }).\text{getEntities:} \\
\quad \quad e \in \text{ <agent definition> } \land e.\text{s-}\text{<type expression>}.\text{s-<base type>} = c.\text{s-<create body>}.\text{s-implicit} >
\]

Mapping to Abstract Syntax

- \[<\text{create request}>(c, \text{exc}) \Rightarrow \text{mk-Graph-node(Mapping(c), Mapping(exc))}> \\
- \[<\text{create body}>(\text{id}, \text{params}) \Rightarrow \text{mk-Create-request-node(Mapping(id), Mapping(params))}> \\

2.8.13.3 Procedure Call

Abstract Syntax

\[
\text{Call-node} :: \text{[ THIS ] Procedure-identifier } \text{ [ Expression ]}^*
\]

Conditions on Abstract Syntax

\[
\forall n \in \text{Call-node} \cup \text{Value-returning-call-node}: \forall d \in \text{Procedure-definition}: \\
(d=\text{getEntityDefinition}(n.\text{s-Procedure-identifier,procedure})) \Rightarrow \\
(isActualAndFormalParameterMatched(n.s-Expression-seq, d.formalParameterSortList),\land \\
(\forall i \in 1..n.s-Expression-seq.length: \\
\quad d.\text{formalParameterList}[i] \in \text{Inout-parameter} \cup \text{Out-parameter} \Rightarrow \\
\quad n.s-Expression[i].\text{in Identifier} \Rightarrow n.s-Expression[i].\text{idKind}=\text{variable}))
\]

The length of the list of optional Expressions must be the same as the number of the Procedure-formal-parameters 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

- \[<\text{procedure call}> :: <\text{procedure call body}> [\text{<on exception>}] > \\
- \[<\text{procedure call body}> :: [ \text{this } ] <\text{procedure}<\text{identifier} > <\text{procedure}<\text{type expression}> ] \ [\text{<actual parameter>}]^* >

Conditions on Concrete Syntax

\[
\forall pc \in <\text{procedure call}>: \\
(let \ apl = pc.s.<procedure call body>.s.<actual parameter>.)\text{seq in} \\
let \ fpl = pc.calledProcedure0.procedureFormalParameterList0 in \\
(fpl.length = apl.length) \land
\]
∀i∈1..fpl.length: 
(fpl[i].parentAS0.parentAS0.s-<parameter kind>∈{\textit{inout}, \textit{out}}) ⇒ 
(\text{apl}[i] \neq \text{undefined}) \land (\text{apl}[i] \in \text{<variable access>∪<extended primary>}) )
endlet)

An <expression> in <actual parameters> corresponding to a formal \textit{in}/\textit{out} or \textit{out} parameter cannot be omitted and must be a <variable access> or <extended primary>.

∀pcd∈<procedure call body>: (pcd.s\ this \neq \text{undefined}) ⇒ 
parentAS0ofKind(pcd, <procedure definition>)=\text{getEntityDefinition}(pcd.s-<identifier>, procedure)

If \textit{this} is used, <procedure identifier> must denote an enclosing procedure.

Transformations

let \textit{nn}=\text{newName} in 
p=\langle\text{procedure call body}\rangle(id, params)
provided parentAS0ofKind(id.s-refersto, <agent type definition>) ≠
parentAS0ofKind(p, <agent type definition>)
=⇒
let par=parentAS0ofKind(p, <agent type definition>) in 
<procedure call body>\langle\textit{id}\langle\textit{fullQualifier}\cap\textit<path item}\langle\textit{entityKind}\cap\textit{entityName}, \textit{nn}, params\rangle\rangle
endlet
and // add the new definition
let \textit{defs}=parentAS0ofKind(p, <agent type definition>).s-<agent structure>.s-<entity in agent>-seq in 
defs \Rightarrow \text{def} \cap
\langle\text{procedure definition}\rangle(\text{empty}, 
\langle\text{procedure heading}\rangle( 
\langle\text{procedure preamble}\rangle(\text{undefined}, \text{undefined}),
\text{empty, nn, empty, undefined, empty, undefined, empty, empty},
\text{empty},
\langle\text{procedure body}\rangle(\text{undefined}, \text{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.

\textit{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) \Rightarrow \text{mk-Graph-node}(\text{Mapping}(p), \text{Mapping}(exc)) |
| <procedure call body>(t, id, params) \Rightarrow \text{mk-Call-node}(\text{Mapping}(t), \text{Mapping}(id), \text{Mapping}(params)) |

2.8.13.4 Output

Abstract Syntax

\begin{tabular}{ll}
\text{Output-node} & :: & \text{Signal-identifier} \\
& & \text{[ Expression ]}* \\
& & \text{[ Signal-destination ]} \\
\text{Signal-destination} & = & \text{Expression} \\
& | & \text{Agent-identifier} \\
& | & \text{THIS} \\
\text{Direct-via} & = & (\text{Channel-identifier} \mid \text{Gate-identifier})-set \\
\end{tabular}
Conditions on Abstract Syntax

\[ \forall n \in \text{Output-node}: \exists d \in \text{Signal-definition}: \\
(d=\text{getEntityDefinition}(n.s, \text{signal})) \land \\
isActualIndFormalParameterMatched(n.s=\text{Expression-seq. d, formalParameterSortList}_1) \]

The length of the list of optional Expressions must be the same as the number of Sort-reference-identifiers 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.

\[ \forall n \in \text{Output-node}: \forall \text{cid} \in \text{Channel-identifier}: (\text{cid} \in \text{Channel-definition}: \exists p \in \text{Channel-path}: \exists g \in \text{Gate-definition}: \\
(d=\text{getEntityDefinition}(\text{cid}, \text{channel})) \land \\
(p, \text{parentAS}=d) \land \\
\text{parentAS}0 \in \text{Kind}(g, \text{Agent-type-definition}) = \text{parentAS}1 \in \text{Kind}(n, \text{Agent-type-definition}) \land \\
\text{g.\text{identifier}}_1 = p.s.\text{Originating-gate} \land \\
(n.s-\text{Signal-identifier} \in g.s-\text{Out-signal-identifier-set}) \land \\
(n.s-\text{Signal-identifier} \in p.s-\text{Signal-identifier-set}) \]

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-identifiers.

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

\[ \forall \text{op} \in \text{output}: \\
(\text{let } cc = \text{op.s} <\text{output body} > ) <\text{communication constraint}> -\text{seq in} \\
(\{ \{ \text{id} \in \text{identifier}: \text{id.\text{kind}} = \text{timer} \in \text{in cc} \} = 0) \land \\
(\{ \text{d} \in \text{destination}: \text{d in cc} \} \leq 1) \land \\
(\{ \text{v} \in \text{via path}: \text{v in cc} \} \geq 0) \]

endlet)

The <communication constraints> in an <output> shall contain no timer <identifier> clause. It contains at most one to <destination> clause and zero or more <via path>s.

\[ \forall \text{op} \in \text{output}: \forall \text{dt} \in \text{destination}:
\\(\text{dt}\in\text{parentAS}0 = \text{op.s} <\text{output body} >) \land (\text{dt}\cdot\text{\text{implicit} = this}) \Rightarrow \\
(\text{op.\text{surroundingScopeUnit}_{is} <\text{agent type definition} >} ) \land \\
(\text{op.\text{surroundingScopeUnit}_{is} \text{\text{surroundingScopeUnit}_{is} <\text{agent type definition} >} ) \\
\]

this may only be specified by an <agent type definition> and in scopes enclosed by an <agent type definition>.

\[ \forall \text{op} <\text{output} > : \forall \text{dt} \in \text{destination}:
\\(\text{dt}\in\text{parentAS}0 = \text{op.s} <\text{output body} >) \land \\
(\text{dt}\cdot\text{\text{implicit} = <\text{expression} >} ) \land (\text{dt}\cdot\text{staticSort} \neq \text{ “Pid”}) \Rightarrow \\
(\text{let } \text{fd} = \text{getEntityDefinitions}(\text{dt}\cdot\text{staticSort}_{is}, \text{interface} ) \text{ in} \\
\forall \text{sig} <\text{identifier}: \text{sig.\text{parentAS}0 = op.s <\text{output body} >} \Rightarrow \\
(\text{sig}\in\text{fd.\text{usedSignalSets}}) \lor (\text{getEntityDefinition}(\text{sig}, \text{signal} ) \in \text{fd.\text{definedSignalSets} } ) \])

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 > \cap 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) =1=>
| 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 `usedSignalSet0` gets the set of signals used in an `<interface definition>`.

```
usedSignalSet0(fd: <interface definition>): SIGNAL0 = def
\{ sig ∈ SIGNAL0:
  (sig∈fd.s-<interface use list>.s-<signal list item>-seq.signalSet0) \lor
  (\exists fd' ∈ <interface definition>: isSubtype0(fd, fd') \land
    (sig∈fd'.s-<interface use list>.s-<signal list item>-seq.signalSet0))\}
```

The function `definedSignalSet0` gets the set of signals defined in an `<interface definition>`.

```
definedSignalSet0(fd: <interface definition>): <signal definition item>-set = def
\{ sd ∈ <signal definition item>:
  ((sd.parentAS0 ∈ <signal definition>) \land (sd.parentAS0.parentAS0 ∈ <entity in interface>) \land
    (sd.parentAS0.parentAS0.parentAS0 = fd)) \lor
  (\exists fd' ∈ <interface definition>: isSubtype0(fd, fd') \land
    (sd.parentAS0.parentAS0 ∈ <signal definition>) \land
    (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

\( \forall dn \in \text{Decision-node}: \forall r, r' \in \text{Range-condition}: \forall ce, ce' \in \text{Constant-expression}:
\)

\( \text{isAncestorAS1}(r, ce) \land \text{isAncestorAS1}(r', ce') \land (ce \neq ce') \land
\)

\( r'. \text{parentAS1} \in dn.\text{s-Decision-answer-set} \land r. \text{parentAS1} \in dn.\text{s-Decision-answer-set} \Rightarrow
\)

\( \text{isCompatibleTo1}(ce, \text{staticSort1}, ce'. \text{staticSort1}) \lor \text{isCompatibleTo1}(ce', \text{staticSort1}, ce. \text{staticSort1}) \land
\)

\( (dn.\text{Decision-question} \in \text{Expression} \Rightarrow \text{isCompatibleTo1}(ce, \text{staticSort1}))
\)

If the Decision-question is an Expression, the Range-condition of the Decision-answers must be sort compatible to the sort of the Decision-question. The Constant-expressions of the Range-conditions must be of a compatible sort.

\( \forall dn \in \text{Decision-node}: \forall r, r' \in \text{Range-condition}:
\)

\( r'. \text{parentAS1} \in dn.\text{s-Decision-answer-set} \land r. \text{parentAS1} \in dn.\text{s-Decision-answer-set} \land
\)

\( r. \text{parentAS1} \neq r'. \text{parentAS1} \Rightarrow r.\text{range1} \cap r'.\text{range1} = \emptyset
\)

The Range-conditions of the Decision-answers must be mutually exclusive.

Concrete Syntax

\( <\text{decision}> :: <\text{question}> \[<\text{on exception}>]\ <\text{decision body}>
\)

\( <\text{decision body}> :: <\text{answer part}>+ \[<\text{else part}>]\n\)

\( <\text{answer part}> :: [<\text{answer}>] \[<\text{transition}>]\n\)

\( <\text{answer}> = <\text{range condition}> | <\text{informal text}>
\)

\( <\text{else part}> :: [<\text{transition}>]
\)

\( <\text{question}> = <\text{expression}> | <\text{informal text}> | \text{any}
\)

Conditions on Concrete Syntax

\( \forall d \in <\text{decision}>: (d.\text{s-<question>} = \text{any}) \Rightarrow
\)

\( \neg(\exists ap \in <\text{answer part}>: (ap.\text{parentAS0}.\text{parentAS0} = d)) \land (ap.\text{s-<answer>} \neq \text{undefined}) \land
\)

\( (d.\text{s-<decision body>} \land .\text{s-<else part>} = \text{undefined})
\)

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

\( <\text{else part}> (\text{undefined}) =1=> <\text{else part}>(<\text{transition gen action statement}>(\text{empty}, \text{undefined}))
\)

\( <\text{answer part}> (a, \text{undefined}) =1=>
\)

\( <\text{answer part}> (a, <\text{transition gen action statement}>(\text{empty}, \text{undefined}))
\)

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).

\( t=<\text{transition gen action statement}>(a, \text{undefined})
\)

\( \text{provided } a.\text{last} \notin <\text{decision}> \land t.\text{parentAS0}.\text{parentAS0} \in <\text{decision}> \land
\)

\( t.\text{findContinueLabel} \neq \text{undefined}
\)

\( =5=> <\text{transition gen action statement}>(a, <\text{terminator statement}>(\text{undefined}, <\text{join}>(\text{findContinueLabel}(t))))
\)

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>.

\( <d=<\text{decision}>(*, *), <\text{action statement}>(\text{undefined}, a) > \text{provided } \Rightarrow \text{TerminatingDecision}(d)
\)

\( =5=> <d, <\text{action statement}>(\text{newName}(%\text{undefined}), a)>
\)
<transition gen action statement>(str, <terminator statement>)(undefined, t)

provided str.last ∈ <decision> ∧ ¬ str.last.TerminatingDecision
=⇒ <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))
=⇒ <decision>(<expression gen primary>(undefined, <any expression>(<identifier>(empty, nn))), exc,
<br decision body>(<answer part>(<expression gen primary>(undefined, idx), ans[idx].s<transition>)
| idx in 1..ans.length, undefined))

and

let parent = d.surroundingScopeUnit0 in
t = <answer part>(ans, trans)
=⇒ mk-Decision-answer(Mapping(ans), Mapping(trans))

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 rangeConditionList0 gets the list of <range condition>s in a <decision>.

rangeConditionList0(d:<decision>):<range condition>* =def

let apl = d.s:<decision body>.s:<answer part>-seq in
apl.answerPartRangeConditionList0
endlet

answerPartRangeConditionList0(apl:<answer part>*):<range condition>* =def
if apl.head.s:<answer> ∈ <range condition> then
apl.head.s:<answer> = apl.tail. answerPartRangeConditionList0
else apl.tail. answerPartRangeConditionList0

The function findContinueLabel computes the continuation label after a decision within a transition string.

findContinueLabel(x: DefinitionAS0): <name> =def
if x ∈ <transition gen action statement> ∧ x.s:<terminator statement> ≠ undefined ∧
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.

\[
\text{TerminatingDecision}(d: \text{<decision>}) : \text{BOOLEAN} = \text{def} \\
(\forall a \in d.s-<answer part>: \text{TerminatingTransition}(a.s-<transition>)) \land \\
(d.s-<else part> = \text{undefined} \lor \text{TerminatingTransition}(d.s-<else part>.s-<transition>)) \\
\]

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**

\[
<\text{statement list}>::= <\text{variable definition statement}>* <\text{statement}>* \\
<\text{statement}>::= <\text{empty statement}> \\
| <\text{compound statement}> \\
| <\text{assignment statement}> \\
| <\text{algorithm action statement}> \\
| <\text{call statement}> \\
| <\text{expression statement}> \\
| <\text{if statement}> \\
| <\text{decision statement}> \\
| <\text{loop statement}> \\
| <\text{terminating statement}> \\
| <\text{labelled statement}> \\
| <\text{exception statement}> \\
<\text{terminating statement}>::= <\text{return statement}> \\
| <\text{break statement}> \\
| <\text{loop break statement}> \\
| <\text{loop continue statement}> \\
| <\text{raise statement}> \\
<\text{variable definition statement}>::= <\text{local variables of sort}>+
\]

**Conditions on Concrete Syntax**

\[
(\forall bs \in <\text{loop break statement}>: \exists ls <\text{loop statement}>: \text{isAncestorAS0}(ls, bs)) \land \\
(\forall cs \in <\text{loop continue statement}>: \exists ls <\text{loop statement}>: \text{isAncestorAS0}(ls, cs))
\]

A <loop break statement> and <loop continue statement> may only occur within a <loop statement>.

\[
\forall ts <\text{terminating statement}> : \forall d \in \text{DefinitionAS0} : \\
\text{ts in } d.s-<\text{statement}~-\text{seq} \Rightarrow ts = d.s-<\text{statement}~-\text{seq}.\text{last}
\]
A `<terminating statement>` may only occur as the last `<statement>` in `<statements>`.

**Mapping to Abstract Syntax**

\[
\begin{align*}
&\langle \langle \text{variable definition statement} \rangle (v \cap \text{rest}) \rangle \quad \text{provided rest \neq empty} =1=> \\
&\langle \langle \text{variable definition statement} \rangle (v) \rangle, \langle \text{variable definition statement} \rangle (\text{rest}) \rangle
\end{align*}
\]

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.

\[
\begin{align*}
&\langle \langle \text{local variables of sort} \rangle (v \cap \text{rest}, s, \text{expr}) \rangle \quad \text{provided rest \neq empty} =1=> \\
&\langle \langle \text{local variables of sort} \rangle (v, s, \text{expr}), \langle \text{local variables of sort} \rangle (\text{rest}, s, \text{expr}) \rangle
\end{align*}
\]

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**

\[
\begin{align*}
&| \langle \text{statement list} \rangle (\text{vars}, \text{allstats} = \text{stats} \cap < \text{terminator }>) \\
&\quad \Rightarrow \text{let nn}=\text{newName in} \\
&\quad \text{mk-Compound-node}(\text{Mapping}(\text{nn}), \text{Mapping}(\text{vars}).\text{toSet}, \text{undefined}, \text{empty}(), \\
&\quad \text{if terminator} \in <\text{terminating statement}> \\
&\quad \text{then mk-Transition}(\text{Mapping}(\text{stats}), \text{Mapping}(\text{terminator})) \\
&\quad \text{else mk-Transition}(\text{Mapping}(\text{allstats}), \text{mk-Break-node}(\text{Mapping}(\text{nn}))) \\
&\quad \text{endif,} \\
&\quad \text{empty})
\end{align*}
\]

\[
\begin{align*}
&| \langle \text{variable definition statement} \rangle (\langle \text{var }\rangle) \Rightarrow \text{Mapping}(\text{var}) \\
&| \langle \text{local variables of sort} \rangle (\langle \text{var }\rangle, s, \text{expr}) \\
&\quad \Rightarrow \text{mk-Variable-definition}(\text{Mapping}(\text{var}), \text{Mapping}(s), \text{Mapping}(\text{expr}))
\end{align*}
\]

**2.8.14.1 Compound Statement**

**Abstract Syntax**

\[
\begin{align*}
\text{Compound-node} &::= \text{Connector-name} \\
\text{Variable-definition-set} &::= [\text{Exception-handler-node}] \\
\text{Init-graph-node}* &::= \text{Transition} \\
\text{Step-graph-node}* &::= \text{Graph-node}
\end{align*}
\]

\[
\begin{align*}
\text{Init-graph-node} &= \text{Graph-node} \\
\text{Step-graph-node} &= \text{Graph-node} \\
\text{Continue-node} &::= \text{Connector-name} \\
\text{Break-node} &::= \text{Connector-name}
\end{align*}
\]

**Concrete Syntax**

\[
\langle \text{compound statement} \rangle ::= \langle \text{statement list} \rangle
\]

**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

```plaintext
<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

```plaintext
∀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.

A `<call statement>` is derived syntax for `<procedure call>` and is transformed into a `<procedure call>` with the same `<procedure call body>`:

```plaintext
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, undefined)>,
  <assignment statement>(<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)>
  )
))

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>.

\[
( \text{<range condition>} ) : \text{task} \{ \text{<statement>-transform} \};
\]

A <decision body> is then formed by taking the transform of each <algorithm answer part> in order and appending the following <else part>.

\[
\text{else} : \text{task} \{ \text{<alternative statement>-transform} \};
\]

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 following <action statement>:

\[
\text{decision ( <expression> ) ;}
\]

Body

enddecision;

Mapping to Abstract Syntax

\[
| \text{<decision statement>(q, <decision statement body>(answers, elseAnswer))} =>
\]

\[
\text{let nn=typeName() in}
\]

\[
\text{mk-Compound-node(Mapping(nn), \emptyset, empty(),}
\]

\[
\text{mk-Transition(}
\]

\[
\text{mk-Decision-node(Mapping(q), undefined,}
\]

\[
\{ \text{mk-Decision-answer(Mapping(a.s=<range condition>),}
\]

\[
\text{if a.s=<statement> \in <terminating statement> then}
\]

\[
\text{mk-Transition(empty(), Mapping(a.s=<statement>))}
\]

\[
\text{elsif a.s=<statement> \in <empty statement> then}
\]

\[
\text{mk-Transition(empty(), mk-Break-node(Mapping(nn)))}
\]

\[
\text{else mk-Transition(< Mapping(a.s=<statement>) >, mk-Break-node(Mapping(nn)))}
\]

\[
\text{endif}
\]

\[
\}
\]

\[
\text{Mapping(elsePart))}
\]

endlet

| <algorithm else part>(trans) => \text{mk-Else-answer(Mapping(trans))}

2.8.14.6 Loop Statements

Concrete Syntax

\[
<\text{loop statement}> ::
\]

\[
<\text{loop clause}>* <\text{loop body statement}> [<\text{finalization statement}>]
\]

<loop body statement> = <statement>

<finalization statement> = <statement>

<loop clause> ::

\[
[<\text{loop variable indication}>] [<\text{Boolean}<expression>] <\text{loop step}>
\]

<loop step> = [<expression> | <procedure call body>]

<loop variable indication> =

\[
<\text{loop variable definition}> | <\text{loop variable indication gen identifier}>
\]

<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

\( \forall ls \in \text{<loop step>}: (\text{let } pd = ls.s.<procedure call body>.calledProcedure_0 \text{ in } pd \neq \text{undefined} <procedure heading>.s.<procedure result> = \text{undefined} \text{ endlet}) \)

The <procedure identifier> in the <procedure call body> of a <loop step> must not refer to a value returning procedure call.

Transformations

\( l = \text{<loop statement>}(*, *, *) \text{ provided } l.\text{parentAS0} \notin <\text{labelled statement}> = 3 = > <\text{labelled statement}>(<\text{label}>(\text{newName}), l) \)

Generate a name for every unlabelled <loop statement>.

\( l = \text{<loop break statement>} = 3 = > <\text{break statement}>(<\text{parentAS0ofKind}(l, <\text{labelled statement}>).s.<\text{name}>) \)

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 mn=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)
\[ \text{mk-Graph-node(mk-Assignment(Mapping(id), Mapping(expr)), undefined)} \]

\[ | <\text{loop clause}>(\text{ind}, *, \text{expr}) \Rightarrow \text{mk-Graph-node(mk-Assignment(} \]
\[ \text{if } c \in <\text{expression}> \text{ then} \]
\[ \text{mk-Graph-node(mk-Assignment(} \]
\[ \text{if } \text{ind} \in <\text{loop variable definition}> \]
\[ \text{then Mapping(ind.s.<name>)} \]
\[ \text{else Mapping(ind.s.<identifier>)} \text{ endif, Mapping(expr)), undefined} \]
\[ \text{else Mapping(c) endif} \]

**Auxiliary Functions**

\[ \text{bigAnd(seq: <expression>*): <expression> } = \text{def} \]
\[ \text{if seq=empty then “True”} \]
\[ \text{elseif seq.length=1 then seq.head} \]
\[ \text{else <operator application>(seq.head, bigAnd(seq.tail))} \]
\[ \text{endif} \]

### 2.8.14.7 Break and Labelled Statements

**Concrete Syntax**

\[ <\text{break statement}> :: <\text{connector}<name> > \]
\[ <\text{labelled statement}> :: <\text{label}> <\text{statement}> \]

**Conditions on Concrete Syntax**

\[ \forall bs\in <\text{break statement}>: \exists ls\in <\text{labelled statement}>: \]
\[ \text{isAncestorAS0}(ls.s.<statement>, bs) \land (bs.s.<name> = ls.s.<label>.s.<name>) \]

A <break statement> must be contained in a statement that has been labelled with the given <connector name>.

**Mapping to Abstract Syntax**

\[ | <\text{break statement}>(\text{name}) \Rightarrow \text{mk-Break-node(Mapping(name))} \]
\[ | <\text{labelled statement}>(\text{name}, \text{body}) \Rightarrow \text{mk-Compound-node(Mapping(name), } \emptyset, \text{ empty,} \]
\[ \text{if } \text{body} \in <\text{terminating statement}> \]
\[ \text{then mk-Transition(} \emptyset(), \text{ Mapping(body))} \]
\[ \text{elsif } \text{body} \in <\text{empty statement}> \]
\[ \text{then mk-Transition(} \emptyset(), \text{ mk-Continue-node(Mapping(nn))} \]
\[ \text{else mk-Transition(< Mapping(body) >, mk-Continue-node(Mapping(nn))} \]
\[ \text{endif, empty}() \) \]

### 2.8.14.8 Empty Statement

**Concrete Syntax**

\[ <\text{empty statement}> :: () \]

**Transformations**

\[ < <\text{empty statement}> > =1=> \text{empty} \]

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

∀ ts ∈ <try statement>: ts ∉ <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

∀n ∈ Set-node ∪ Reset-node: ∀d ∈ Timer-definition:
(d=getEntityDefinition,(n,s-Timer-identifier,timer)) ⇒
isActualAndFormalParameterMatched(n,s-Expression-seq, d.formalParameterSortList)

The sorts of the list of Expressions in the Set-node and Reset-node must correspond by position to the list of Sort-reference-identifiers 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)  
=⇒ <set clause>({operator application}("+", now, id.refersto0, s-<timer default initialization>), id, exprList)

A <set clause> with no <Time expression> is derived syntax for a <set clause> where <Time expression> is now + <Duration constant expression>

where <Duration constant expression> is derived from the <timer default initialization> in timer definition.

< <set>(< s > ∩ rest) > provided rest ≠ empty =⇒  
< <set>(< s >), <set>(rest) >

< <reset>(< r > ∩ rest) > provided rest ≠ empty =⇒  
< <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 > ∩ rest) > provided rest ≠ empty =⇒  
< <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)) |
2.8.16 Exception

2.8.16.1 Exception Definition

Abstract Syntax

\[
\text{Exception-definition} :: \text{Exception-name} \\
\quad \quad \quad \quad \text{Sort-reference-identifier}^* 
\]

Concrete Syntax

\[
<\text{exception definition}> ::= <\text{exception definition item}>+ \\
<\text{exception definition item}> ::= \\
\quad <\text{name}> <\text{sort}>^* 
\]

Transformations

\[
<\text{exception definition}>(<t \cap \text{rest}) > \text{provided rest ≠ empty }=1=> \\
<\text{exception definition}>(<s >), <\text{exception definition}>(<\text{rest}> \\
\]

An \text{exception definition} may contain several \text{exception definition item}s. This is derived syntax for specifying a sequence of \text{exception definitions}s, one for each \text{exception definition item}. This is an auxiliary transformation.

Mapping to Abstract Syntax

\[
|<\text{exception definition}>(<\text{item}>) => Mapping(\text{item}) \\
|<\text{exception definition item}>(\text{name}, \text{sortList}) => \text{mk-Exception-definition}(Mapping(\text{name}), Mapping(\text{sortList})) 
\]

2.8.16.2 Exception Handler

Abstract Syntax

\[
\text{Exception-handler-node} :: \text{Exception-handler-name} \\
\quad \quad \quad \quad [ \text{On-exception} ] \\
\quad \quad \quad \quad \text{Handle-node-set} \\
\quad \quad \quad \quad [ \text{Else-handle-node} ] 
\]

Conditions on Abstract Syntax

\[
\forall n, n' \in \text{Exception-handler-node}: \\
\quad \text{(parentAS1ofKind}(n, \text{State-transition-graph} \cup \text{Procedure-graph})= \\
\quad \text{parentAS1ofKind}(n', \text{State-transition-graph} \cup \text{Procedure-graph})) \cap n \neq n' \Rightarrow \\
\quad n.\text{s-Exception-handler-name} \neq n'.\text{s-Exception-handler-name} 
\]

The \text{Exception-handler-nodes} within a given \text{State-transition-graph} or \text{Procedure-graph} must all have different \text{Exception-handler-name}.

Concrete Syntax

\[
<\text{exception handler}> :: \\
\quad <\text{exception handler list}> [ \langle \text{on exception} \rangle ] <\text{handle}>^* \\
\quad <\text{exception handler list}> = 
\]
Conditions on Concrete Syntax

\[ \forall eh \in \text{exception handler list}: eh.s - \text{exception handler list} : s - \Rightarrow \text{seq in} \]

The \text{exception handler name}s in an \text{asterisk exception handler list} must be distinct and must be contained in other \text{exception handler list}s in the enclosing body or in the body of a supertype.

\[ \forall eh1, eh2 \in \text{exception handler list}: eh1 \text{.surroundingScopeUnit} = eh2 \text{.surroundingScopeUnit} \Rightarrow \]

An \text{exception handler} has at most one \text{exception handler} associated.

Transformations

When several \text{exception handler}s contain the same \text{exception handler name}, these \text{exception handler}s are concatenated into one \text{exception handler} having that \text{exception handler name}.

Mapping to Abstract Syntax

| \text{exception handler}(<name >), exc, handles, * )

Auxiliary Functions

The function \text{exceptionHandlerSet}_0 gets the set of \text{exception handler}s defined in the given definition.

\[ \text{exceptionHandlerSet}_0(d: \text{agent type definition}) \cup \text{agent definition} \cup \text{procedure definition} \cup \]

\[ \text{composite state type definition} \cup \text{composite state} \cup \]

\[ \text{operation definition} : \text{exception handler} \rightarrow \text{set} \]

\[ d \text{.localExceptionHandlerSet}_0.d \text{.inheritedExceptionHandlerSet}_0 \]
\[
localExceptionHandlerSet_0(d) = (\text{<agent type definition>} \cup \text{<agent definition>} \cup \text{<procedure definitions>} \cup \\
\text{<composite state type definition>} \cup \text{<composite state>} \cup \\
\text{<operation definition>}) = \text{<exception handler>-set} = \{ eh \in \text{<exception handler>}: eh\text{-surroundingScopeUnit}_0 = d \} \\
\]

\[
\text{inheritedExceptionHandlerSet}_0(d) = (\text{<agent type definition>} \cup \text{<agent definition>} \cup \\
\text{<procedure definitions>} \cup \text{<composite state type definition>} \cup \text{<composite state>} \cup \\
\text{<operation definition>}) = \text{<exception handler>-set} = \{ \text{if (d.specialization}_0=\text{undefined) then } \emptyset \\
\text{else d.specialization}_0.s\text{-<type expression>.baseType}_0\text{-exceptionHandlerSet}_0 \} \\
\]

The functions \text{associatedExceptionHandler}_0 gets the \text{<exception handler>}s associated with a exception handler name.

\[
\text{associatedExceptionHandler}_0(hn: \text{<name>}): \text{<exception handler> =def}
\text{take(}\{ eh \in \text{<exception handler>: hn} \in eh\text{-exceptionHandlerNameSet}_0\}) \\
\]

### 2.8.16.3 On-Exception

**Abstract Syntax**

\[
\text{On-exception} :: \text{Exception-handler-name} \\
\]

**Conditions on Abstract Syntax**

\[
\forall oe \in \text{On-exception}: \exists hn \in \text{Exception-handler-node}: \\
(oe.s\text{-Exception-handler-name} = hn.s\text{-Exception-handler-name}) \land \\
\text{(parentAS1ofKind}(oe, \text{State-transition-graph} \cup \text{Procedure-graph})= \\
\text{parentAS1ofKind}(hn, \text{State-transition-graph} \cup \text{Procedure-graph})) \\
\]

The \text{Exception-handler-name} specified in \text{On-exception} must be the name of an \text{Exception-handler-node} within the same \text{State-transition-graph} or \text{Procedure-graph}.

**Concrete Syntax**

\[
<\text{on exception}> :: <\text{exception handler}<\text{name}> \\
\]

**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 \text{<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**

\[
\mid <\text{on exception}><\text{name}> \Rightarrow \text{mk-On-exception}(\text{Mapping}(\text{name})) \\
\]

### 2.8.16.4 Handle

**Abstract Syntax**

\[
\text{Handle-node} :: \text{Exception-identifier} \\
[ \text{Variable-identifier} ]^* \\
[ \text{On-exception} ] \\
\text{Transition} \\
\]

\[
\text{Else-handle-node} :: [ \text{On-exception} ] \text{Transition} \\
\]
Conditions on Abstract Syntax

∀hn, hn' ∈ Handle-node:

hn. parentASI = hn'. parentASI ∧ hn.s-Exception-identifier = hn'.s-Exception-identifier ⇒ hn = hn'

The Exception-identifiers in the Handle-node-set must be distinct.

∀hn ∈ Handle-node:

(let d = getEntityDefinition((hn.s-Exception-identifier, exception) in
isActualAndFormalParameterMatched((hn.s-Variable-identifier-seq, d.formalParameterSortList))
endlet)

The length of the list of optional Variable-identifiers in Handle-node must be the same as the number of Sort-reference-identifiers 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, < <exception stimulus>(id, vars) >, exc, <transition gen action statement>(actions, term)) >

provided rest ≠ empty =⇒
< <handle>(virt, < <exception stimulus>(id, vars) >, exc, <transition gen action statement>(actions, term)) >

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) \} ≠ ∅ =⇒

<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>* (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

\[
\text{<data definition>} = \\
\text{<data type definition>} | \text{<interface definition>} | \text{<syntype definition>} | \text{<synonym definition>}
\]

\[
\text{<sort>} = \text{<basic sort>} | \text{<anchored sort>} | \text{<expanded sort>} | \text{<reference sort>} | \text{<pid sort>}
\]

\[
\text{<basic sort>} = \text{<sort>} \langle \text{<identifier>} | \text{<syntype}> \rangle
\]

\[
\text{<anchored sort>} :: [\text{<basic sort>}]
\]

\[
\text{<expanded sort>} :: \text{<basic sort>}
\]

\[
\text{<reference sort>} :: \text{<basic sort>}
\]

\[
\text{<pid sort>} = \text{<sort>} \langle \text{<identifier>} \rangle
\]

Transformations

\[
\text{<expanded sort>} (\text{sort}) \text{ provided } \neg \text{ sort.isObjectSort}_0 \\
\Rightarrow 8 \Rightarrow \text{ sort}
\]

An <expanded sort> with a <basic sort> that represents a value sort is replaced by the <basic sort>.

\[
\text{<expanded sort>} (\text{sort}) \text{ provided } \text{ sort.isObjectSort}_0 \\
\Rightarrow 8 \Rightarrow \text{ sort A <reference sort> with a <basic sort> that represents an object sort is replaced by the <basic sort>.}
\]

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

\[
| \text{<expanded sort>} (\text{ident}) \Rightarrow \text{ mk-Expanded-sort} (\text{Mapping} (\text{ident}))
\]

\[
| \text{<reference sort>} (\text{ident}) \Rightarrow \text{ mk-Reference-sort} (\text{Mapping} (\text{ident}))
\]

2.9.2 Data Type Definition

Abstract Syntax

\[
\text{Data-type-definition} = \text{ Value-data-type-definition} \\
\text{ Object-data-type-definition} \\
\text{ Interface-definition}
\]

\[
\text{Value-data-type-definition} :: \text{ Sort} \\
\text{ Data-type-identifier} \\
\text{ Literal-signature-set} \\
\text{ Static-operation-signature-set} \\
\text{ Dynamic-operation-signature-set} \\
\text{ Data-type-definition-set} \\
\text{ Syntype-definition-set} \\
\text{ Exception-definition-set}
\]

\[
\text{Object-data-type-definition} :: \text{ Sort} \\
\text{ Data-type-identifier}
\]
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

∀ dtd ∈ <data type definition>: ∀ fcp ∈ <formal context parameter>:
fcp ∈ dtd.localFormalContextParameterList.toSet ⇒
fcp ∈ <sort context parameter> ∪ <synonym context parameter>

A <formal context parameter> of <formal context parameters> must be either a <sort context parameter> or a <synonym context parameter>.

∀ anchSort ∈ <anchored sort>:
parentAS0ofKind(anchSort, <data type definition>) ≠ undefined ∧ (anchSort.s <basic sort> ≠ undefined ⇒
parentAS0ofKind(anchSort, <data type definition>) =
getEntityDefinition(anchSort.s <basic sort>, sort))

An <anchored sort> is legal concrete syntax only if it occurs within a <data type definition>. The <basic sort> in the <anchored sort> must name the <sort> introduced by the <data type definition>.

∀ sid ∈ <pid sort> : isPidSort(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 > ∧
< <interface definition> (undefined, <interface heading> (name, <<, undefined),
<interface specialization> (1
< <type expression> (<identifier> (<path item> (<system type> name), name)) ), <<, undefined) >

<s = <block type definition> (*, <block type heading> (name, heading))) >
= 9 =>
<s > ∧
< <interface definition> (undefined, <interface heading> (name, <<, undefined),
<interface specialization> (1
< <type expression> (<identifier> (<path item> (<block type> name), name)) ), <<, undefined) >
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, *, *)) => 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, *, *)) => mk-Value-data-type-definition(Mapping(name), Mapping(base),}
\{ e \in \text{Mapping(entities).toSet}: e \in \text{Literal-signature} \},
\{ e \in \text{Mapping(entities).toSet}: e \in \text{Static-operation-signature} \},
\{ e \in \text{Mapping(entities).toSet}: e \in \text{Dynamic-operation-signature} \},
\{ e \in \text{Mapping(entities).toSet}: e \in \text{Data-type-definition} \},
\{ e \in \text{Mapping(entities).toSet}: e \in \text{Syntype-definition} \},
\{ e \in \text{Mapping(entities).toSet}: e \in \text{Exception-definition} \}\)

\[
| \text{<data type definition body>}(\text{entities, ctor, operations, *}) => \text{Mapping(entities)} \cap \text{Mapping(ctor)}
\]

\[
| \text{<operations>}(*, \text{bodies}) => \text{Mapping(bodies)}
\]

**Auxiliary Functions**

The function \(\text{localDataTypeDefinitionSet}_0\) gets the <data type definition>s defined locally in a scope unit.

\[
\text{localDataTypeDefinitionSet}_0(\text{su}: \text{SCOPEUNIT}_0): \text{<data type definition>-set} = \text{def}
\{ \text{de} \in \text{<data type definition>}: \text{de}.\text{surroundingScopeUnit}_0 = \text{su} \}
\]

The function \(\text{impliedBases}\) computes the implied base interface.

\[
\text{impliedBases}(g: \text{<gate in definition>* } \cup \text{<gate constraint> } \cup \text{<signal list item> }): \text{<type expression>+} = \text{def}
\text{case } g \text{ of}
| \text{<textual gate definition>}(*, \text{c1, c2}) > \cap \text{tail} =>
\quad \text{impliedBases(c1)} \cap \text{impliedBases(c2)} \cap \text{impliedBases(tail)}
| \text{<textual interface gate definition>}(\text{*, ident}) > \cap \text{tail} =>
\quad < \text{ident} > \cap \text{impliedBases(tail)}
| \text{<gate constraint>}(\text{in}, *, \text{signals}) => \text{BigSeq(\text{impliedBases(s)} | s in \text{signals})}
| \text{<signal list item>}(\text{interface}, \text{ident}) => < \text{ident} >
| \text{otherwise} \quad \text{empty}
\text{endcase}
\]

The function \(\text{gatesignalset0}\) computes the in signals of a gate list.

\[
\text{gatesignalset0}(g: \text{<gate in definition>* } \cup \text{<gate constraint> } \cup \text{<signal list item> }): \text{SIGNAL}_0 = \text{def}
\text{case } g \text{ of}
| \text{<textual gate definition>}(*, \text{c1, c2}) > \cap \text{tail} =>
\quad \text{gatesignalset0(c1)} \cup \text{gatesignalset0(c2)} \cup \text{gatesignalset0(tail)}
| \text{<textual interface gate definition>}(\text{*, ident}) > \cap \text{tail} =>
\quad \text{let} \quad \text{fd} = \text{getEntityDefinition}(\text{ident, interface}) \text{ in}
\quad \text{fd.usedSignalSet}_0 \cup \{ \text{sd}.\text{identifier}_0: \text{sd} \in \text{fd.definedSignalSet}_0 \} \cup \text{tail.gatesignalset0}
| \text{<gate constraint>}(\text{in}, *, \text{signals}) => \text{signals.signalSet}_0
| \text{otherwise} \quad \emptyset
\text{endcase}
\]

### 2.9.3 Interface Type

**Abstract Syntax**

\[
\text{Interface-definition} :: = \text{Sort}
\quad \text{Data-type-identifier*}
\quad \text{Signal-definition-set}
\quad \text{Exception-definition-set}
\]

\[
\text{Sort} = \text{Name}
\]
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

∀fd∈<interface definition>: isRestrictedByInterface₀(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.

∀ind∈<interface definition>: ∀fcp∈<formal context parameter>:
   fcp∈ind.localFormalContextParameterList₀ ⇒
   fcp∈<signal context parameter> ∪<remote procedure context parameter> ∪<remote variable context parameter> ∪<sort context parameter> ∪<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∈Mapping(entities).toSet: (e∈Signal-definition)}
| {} e∈Mapping(entities).toSet: (e∈Exception-definition)}

Auxiliary Functions

The function localInterfaceDefinitionSet₀ gets the local defined interface definition list of a scope unit.

localInterfaceDefinitionSet₀(su: SCOPEUNIT₀): <interface definition>-set=def
   {de∈<interface definition>:d.surroundingScopeUnit₀=su}

The function isRestrictedByInterface₀ decides if a <signal list> only contains <signal identifier>s, <remote procedure identifier>s, <remote variable identifier>s and <signal list identifiers>. If a <signal list identifier> is part of the <signal list> it must also respect this restriction.

isRestrictedByInterface₀(sl: <signal list item>*): BOOLEAN =def
   if sl=empty then True
   else
      case sl.head.s-implicit of
         | <signal, remote procedure, remote variable> => isRestrictedByInterface₀(sl.tail)
         | (signallist) =>
            let sl'=getEntityDefinitionₙ(sl.head.s⟨<identifier>, signallist⟩.s⟨<signal list item>-seq⟩ in
            isRestrictedByInterface₀(sl')∧isRestrictedByInterface₀(sl.tail)

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
<rename pair gen literal name> :: <literal name> <base type

<interface specialization> :: <interface
<type expression>+ Conditions on Concrete Syntax

∀dataDef ∈<data type definition>:∀superTypeDef ∈<data type definition>:
isSubtype0(dataDef, superTypeDef) ⇒
superTypeDef.<data type definition body>.s-.<data type constructor>=undefined ∨
isSameConstructorKind(dataDef.<data type definition body>.s-.<data type constructor>,
superTypeDef.<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 expression> in the <data type specialization>.

∀rn ∈<renaming>:
let lnl = <<rp.<literal name>, rp.s2.<literal name>> | rp in rn.s.<rename pair>-seq > in
let onl = <<rp.<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.<type expression>.<base Type0 in
let onl = <rp.s2.<operation name> | rp in sp. s.<rename pair>-seq: rp ∈<rename pair gen operation name>= in
∀one <operation name> : on in onl⇒
(∃os ∈<operation signature> : os.surroundingScopeUnit0 = bt ∧ on = os.name0)
endlet)

A <base type operation name> specified in a <rename list> must be an operation with <operation name> defined in the data type definition defining the <base type> of <data type 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>(<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) \cap \text{inheritedOperations(spec).s.<operation signatures>.s.<operator list>}, 
    <method list>(meths) \cap \text{inheritedOperations(spec).s.<operation signatures>.s.<method list>), refs, def(s), 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 \(\text{isRenamedBy}_{\text{spec}}\) determine if a <literal signature> or an <operation signature> is renamed by a <specialization>.

\[
\text{isRenamedBy}_{\text{spec}}(sn:<\text{literal signature}>, spec:<\text{specialization}>): \text{BOOLEAN} = \exists rp \in <\text{rename pair}>:\ (
\text{rp. parentAS0, parentAS0} = spec) \land
\text{rp.s2.<literal name> = sn.name}_{\text{spec}} \lor \text{rp.s2.<operation name> = sn.name}_{\text{spec}}
\]

The function \(\text{isSameConstructorKind}_{\text{c1}, \text{c2}}\) is used to determine if two data type constructor is of the same kind.

\[
\text{isSameConstructorKind}_{\text{c1}, \text{c2}}(\text{c1:<data type constructor>}, \text{c2:<data type constructor>}): \text{BOOLEAN} = \exists c1 \in <\text{literal list}> \lor \exists c2 \in <\text{structure definition}> \lor \exists c1 \in <\text{choice definition}>
\]

Sort compatibility determines when a sort can be used in place of another sort, and when it cannot. The function \(\text{isSortCompatible}_{\text{sort1}, \text{sort2}}\) is used to determine if the first sort is sort compatible to the second one.

\[
\text{isSortCompatible}_{\text{sort1}, \text{sort2}}(\text{sort1}, \text{sort2}): \text{BOOLEAN} = \exists s_{\text{sort1}}(\text{sort1}, \text{sort2}) \lor \text{isDirectlySortCompatible}_{\text{sort1}, \text{sort2}} \lor ((\text{isObjectSort}_{\text{sort2}} \lor \text{isPidSort}_{\text{sort2}}) \land
\]

The function isSameSort₀ is used to determine if the given two sorts are the same.

\[
\exists \text{sort1} \in \text{<sort>}: \text{isSortCompatible}(\text{sort1}, \text{sort3}) \land \text{isSortCompatible}(\text{sort3}, \text{sort2})
\]

The function isSameSort₀ is used to determine if the sort in the first argument is directly sort compatible to the one in the second.

\[
isSameSortₐ(\text{sort1}: \text{<sort>}, \text{sort2}: \text{<sort>}): \text{BOOLEAN} =_{\text{def}}
\]

\[
\begin{aligned}
sort1 &= \text{sort2} \lor \\
(&\text{sort1} \in \text{<basic sort>} \land \text{sort2} \in \text{<basic sort>} \land \\
&\text{getEntityDefinition}(\text{sort1}, \text{sort}) \text{.derivedDateType} = \text{getEntityDefinition}(\text{sort2}, \text{sort}) \text{.derivedDateType}) \lor \\
&(\text{sort1} \in \text{<anchored sort>} \land \text{sort2} \in \text{<anchored sort>} \land \\
&\text{parentAS0ofKind}(\text{sort1}, \text{<data type definition>}) = \text{parentAS0ofKind}(\text{sort2}, \text{<data type definition>})) \lor \\
&(\text{sort1} \in \text{<expanded sort>} \land \text{sort2} \in \text{<expanded sort>} \land \\
&\text{getEntityDefinition}(\text{sort1}, \text{<type>}) \text{.derivedDateType} = \\
&\text{getEntityDefinition}(\text{sort2}, \text{<type>}) \text{.derivedDateType}) \lor \\
&(\text{sort1} \in \text{<reference sort>} \land \text{sort2} \in \text{<reference sort>} \land \\
&\text{getEntityDefinition}(\text{sort1}, \text{<type>}) \text{.derivedDateType} = \\
&\text{getEntityDefinition}(\text{sort2}, \text{<type>}) \text{.derivedDateType}) \lor \\
&(\text{sort1} \in \text{<pid sort>} \land \text{sort2} \in \text{<pid sort>} \land \\
&\text{getEntityDefinition}(\text{sort1}, \text{sort}) = \text{getEntityDefinition}(\text{sort2}, \text{sort})
\end{aligned}
\]

Determine if two sort lists are the same.

\[
isSameSortListₐ(\text{sl, sl'}, \text{<sort>*}): \text{BOOLEAN} =_{\text{def}}
\]

\[
(\forall i \in 1..\text{sl.length}: \text{isSameSort₀}(\text{sl}[i], \text{sl'}[i]))
\]

The function isObjectSort₀ is used to determine if a sort is an object sort.

\[
isObjectSortₐ(\text{sort}: \text{<sort>}): \text{BOOLEAN} =_{\text{def}}
\]

\[
\begin{aligned}
\text{case} \text{sort} \text{of} \\
| \text{<basic sort>} => \\
&\text{let} \text{did} = \text{getEntityDefinition}(\text{sort}, \text{type}) \text{.derivedDateType} \text{in} \\
&\text{if} \text{did.}\text{s} \text{<data type heading}.s \text{<data type kind} = \text{object then True} \\
&\text{else False} \\
&\text{endif} \\
| \text{<anchored sort>} => \text{isObjectSort₀}(\text{sort.}\text{s} \text{<basic sort>}) \\
| \text{<expanded sort>} => \text{True} \\
\text{otherwise False}
\end{aligned}
\]

The function isPidSort₀ is used to determine if a sort is an pid sort.

\[
isPidSortₐ(\text{sort}: \text{<sort>}): \text{BOOLEAN} =_{\text{def}}
\]

\[
\text{getEntityDefinition}(\text{sort,sort}) \in \text{<interface definition>}
\]

The function isSubSort₀ is used to determine if the sort given in the first argument is a super sort of the one in the second.

\[
isSubSortₐ(\text{sort1}: \text{<sort>}, \text{sort2}: \text{<sort>}): \text{BOOLEAN} =_{\text{def}}
\]

\[
\begin{aligned}
&\text{let} \text{td1} = \text{getEntityDefinition}(\text{sort1}, \text{sort}) \text{in} \\
&\text{let} \text{td2} = \text{getEntityDefinition}(\text{sort2}, \text{sort}) \text{in}
\end{aligned}
\]
(td1 ∈ <interface definition> ⇒ isSubtype0(td1, td2)) ∧
(td1 ∈ <data type definition> ∪ <syntype definition> ⇒
isSubtype0(td1.derivedDateString, td2.derivedDateString))
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)
  else undefined endif
otherwise undefined endif
endlet

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

∀ os ∈ <operation signature> :
  os.parentAS0 ∈ <operator list> ⇒ (∃ v ∈ <virtuality> U <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>.

∀ os ∈ <operation signature> :
  (os.parentAS0 ∈ <method list> ∧ os.virtuality = undefined) ⇒
  (∃ av ∈ <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>.

∀ os ∈ <operation signature> : os.<$result> = undefined ⇒ os.parentAS0 ∈ <method list>

<result> in <operation signature> may be omitted only if the <operation signature> occurred in a <method list>.

∀ os ∈ <operation signature> :
  (∃ av ∈ <argument virtuality> : av.parentAS0.parentAS0 = os) ⇒ os.virtuality ∈ { 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>) \( \cap \) rest)
=⇒
  <operation signatures>(<operator list>(operations \( \cap \)
    <operation signature>(<pre>, name,
      <argument>((<argument virtuality>(), parentAS0ofKind(<o, SCOPEUNIT0).identifier0),
      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>.
If the \textit{sort} of an \textit{argument} is an \textit{anchored sort}, the \textit{argument} implicitly contains \textit{argument virtuality}.

\begin{verbatim}
s = <operation signature>(p = <operation preamble>(redefined, *), name, args, result, raises)
=2=>
<operation signature>(p, name, updateVirtuality(s.superCounterparts, args), result, raises)
\end{verbatim}

If an \textit{operation signature} contains the keywords \texttt{redefined} in \textit{virtuality}, for every \textit{argument} in the matching \textit{operation signature} of the base type, if this \textit{argument} (implicitly or explicitly) contains \textit{argument virtuality}, then the corresponding \textit{argument} in \textit{operation signature} implicitly also contains \textit{argument virtuality}.

\begin{verbatim}
<formal parameter>(undefined, sort) =2=> <formal parameter>(in, sort)
\end{verbatim}

An \textit{argument} without an explicit \textit{parameter kind} has the implicit \textit{parameter kind} \texttt{in}.

### Mapping to Abstract Syntax

| \textit{operation signatures}(operators, *) => Mapping(operators).toSet |
| os = <operation signature>(*, name, arguments, result, *) => |
| if \{ v ∈ <virtuality>:\texttt{argument virtuality}: isAncestorAS0(os, v) \} = empty then |
| \texttt{mk-Static-operation-signature}(Mapping(name), Mapping(arguments), Mapping(result), Mapping(os.operatorProcedureName)) |
| else |
| \texttt{mk-Static-operation-signature}(Mapping(name), Mapping(arguments), Mapping(result), Mapping(os.operatorProcedureName)) |
| endif |

| <argument>(undefined, <formal parameter>(*, name)) => \texttt{mk-Nonvirtual-argument}(Mapping(name)) |
| <argument>(*, <formal parameter>(*, name)) => \texttt{mk-Virtual-argument}(Mapping(name)) |

### Auxiliary Functions

The function \texttt{operatorProcedureName} associates each operation signature with its implicit anonymous procedure name.

\begin{verbatim}
controlled operatorProcedureName: <operation signature> → <identifier>
initially ∀ o ∈ <operation signature> : o.operatorProcedureName = empty
\end{verbatim}

Get the list of the arguments of an operation signature.

\begin{verbatim}
operationSignatureParameterList(os: <operation signature>):<argument>* =def
(os.s-<argument>-seq)
\end{verbatim}

Update the virtuality in the arguments list of the derived operation signature given the argument list of the base signature.

\begin{verbatim}
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)
\end{verbatim}

### 2.9.6 Any

#### Transformations

\begin{verbatim}
<data type definition>(use, preamble, heading, undefined, body) =1=>
<data type definition>(use, preamble, heading,
<data type specialization>(<identifier>(<> , <name>("Any")),<renaming>(<>)))
\end{verbatim}
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

\[\text{Literal-signature} = \text{Single-literal} \mid \text{Nameclass-literal}\]

\[\text{Single-literal} :: \text{Literal-name} \mid \text{Name.class literal} \mid \text{Named number}\]

\[\text{Named number} :: \text{Literal-name} \mid \text{Natural simple expression}\]

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

\[\forall \text{num1, num2} \in \text{Named number}: \text{num1.parentAS0=num2.parentAS0} \land \text{num1=num2} \Rightarrow \text{num1.s-value_0} \neq \text{num2.s-value_0}\]

Each result of <Natural simple expression> occurring in a <named number> must be unique among all <literal signature>s in the <literal list>.

Transformations

\[<\text{literal list}> (\text{head} \cap <\text{name}> \cap <\text{tail}>)
\text{provided name} \in <\text{literal name}>
\land \forall n \in <\text{literal name}> : \neg n \in \text{head}
\Rightarrow
<\text{literal list}> (\text{head} \cap <\text{named number}> (\text{name}, \text{nextNumber(head)}) \cap <\text{tail}>)
\]

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.

\[b = \text{<data type definition body>} (\text{entities}, s = <\text{literal list}>(*, *), <\text{operations}> (\text{<operation signatures}> (<\text{operator list}> (\text{operators}), \text{refs}, \text{defs}), \text{init}))
\text{provided} \neg b.\text{parentAS0.identifier_0.implicitSignaturesAdded}
\Rightarrow
\text{let sort} = b.\text{parentAS0.identifier_0 in}
\text{let newoperators} = <\text{operation signature}> (\text{<operation preamble>} (\text{undefined, public}), <name> ("\text{\textless}")),
<\text{argument}> (\text{undefined, <formal parameter>} (\text{in, <anchored sort}> (\text{sort}))),
<\text{argument}> (\text{undefined, <formal parameter>} (\text{in, <anchored sort}> (\text{sort}))),
\langle\text{result}> (\text{<identifier>} (\text{<qualifier>} (\text{<name>} ("\text{Predefined}")), "\text{Boolean}")), \text{\textgreater}angle),
<\text{operation signature}> (\text{<operation preamble>} (\text{undefined, public}), <name> ("\text{\textgreater}")),
<\text{argument}> (\text{undefined, <formal parameter>} (\text{in, <anchored sort}> (\text{sort}))),
<\text{argument}> (\text{undefined, <formal parameter>} (\text{in, <anchored sort}> (\text{sort}))),\]
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;
suc  (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, gets the <visibility> part for an <operation signature> or a <literal signature>.

visibility(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
else 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

\[
<\text{structure definition}> :: [\text{<visibility>}] <\text{field}>^* \\
<\text{field}> = \\
\text{<optional field>} | \text{<mandatory field>} \\
\text{<optional field>} :: <\text{fields of sort}> \\
\text{<mandatory field>} :: <\text{fields of sort}> [\text{<field default initialization>}] \\
\text{<field default initialization>} :: \text{<constant expression>} \\
\text{<fields of sort>} :: [\text{<visibility>}] <\text{name}>^+ <\text{sort}> 
\]

Conditions on Concrete Syntax

\[\forall s\in <\text{structure definition}>: s.\text{fieldNameList}_0.\text{length} = |s.\text{fieldNameList}_{0..}| \]

Each <field name> of a structure sort must be different from every other <field name> of the same <structure definition>.

Transformations

\[
<\text{<optional field>}(<\text{fields of sort}>(\text{vis}, <f> \land \text{rest}, \text{sort}))) \text{ provided } \text{rest} \neq \text{empty} =_{1=1} \Rightarrow \]
\[
<\text{<optional field>}(<\text{fields of sort}>(\text{vis}, <s>, \text{sort})), <\text{field default initialization}>) \\
\text{<mandatory field>} :: <\text{fields of sort}> (\text{vis}, <s>, \text{sort}), \text{init}) \text{ provided } \text{rest} \neq \text{empty} =_{1=1} \Rightarrow \]
\[
<\text{<mandatory field>}(<\text{fields of sort}>(\text{vis}, <s>, \text{sort}), \text{init}), <\text{mandatory field}>(<\text{fields of sort}>(\text{vis}, \text{rest}, \text{sort}), \text{init})) \\
\]

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.

\[
b = <\text{data type definition body}>(\text{entities}, s = <\text{structure definition}>(*, \text{fields}), <\text{operations}> (<\text{operation signatures}>(<\text{operator list}>(<\text{operators}>), <\text{method list}>(<\text{methods}>), \text{refs}, \text{defs}), \text{init}))) \text{ provided } \sim \]
\[
\text{let } \text{sort} = b.\text{parentAS0}.\text{identifiers}.\text{implicitSignaturesAdded}\n\text{let newoperators} = \]
\[
< \text{operation signature}>(<\text{operation preamble}>(\text{undefined}, \text{public}), \text{name}("\text{Make}")), \text{<argument>(undefined, <formal parameter>)(in}s_1)) | s \text{ in } s.\text{fieldSortList}_p, \text{result}(\text{sort}, <>)) \text{ in } 1.\text{fields.\text{NameList}_0.length} > \land \]
\[
< \text{operation signature}>(<\text{operation preamble}>(\text{virtual}, \text{public}), \text{<name>(fields.\text{NameList}_0[n].s-TOKEN + "Modify"), <argument>(undefined, <formal parameter>)(in} fields.\text{fieldSortList}_0[n])) >, \text{result}((\text{sort}, <>)) | n \text{ in } 1.\text{fields.\text{NameList}_0.length} > \land \]
\[
< \text{operation signature}>(<\text{operation preamble}>(\text{virtual}, \text{public}), \text{<name>(fields.\text{NameList}_0[n].s-TOKEN + "Extract")}, >, \text{result}((\text{fields.\text{SortList}_0[n]}, <>)) | n \text{ in } 1.\text{fields.\text{NameList}_0.length} > \land 
\]
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.

If a <field> is defined with a <field default initialization>, this is derived syntax for the definition of this <field> as optional.
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 \textit{constant expression} after structure creation is added.

\textbf{Auxiliary Functions}

The function \texttt{fieldNameList} retrieves the list of field names defined in a \texttt{structure definition} or a \texttt{choice definition}.

\begin{verbatim}
fieldNameList(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
\end{verbatim}

The function \texttt{fieldSortList} gets the list of field sorts defined in a \texttt{structure definition} or a \texttt{choice definition}.

\begin{verbatim}
fieldSortList(sd:<structure definition>∪<choice definition>):<sort>*=def
  <fn.parentAS0.s=<sort> | fn in sd.fieldNameList>isOptionalField(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
\end{verbatim}

\section*{2.9.7.3 Choice Data Types}

\textbf{Concrete Syntax}

\begin{verbatim}
<choice definition> :: [visibility] <choice of sort>*
<choice of sort> :: [visibility] <field><name> <sort>
\end{verbatim}

\textbf{Conditions on Concrete Syntax}

\begin{verbatim}
∀cd∈<choice definition>: cd.fieldNameList0.length = |cd.fieldNameList0.toSet|
\end{verbatim}

Each \texttt{field name} of a choice sort must be different from every other \texttt{field name} of the same \texttt{choice definition}.

\textbf{Transformations}

\begin{verbatim}
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))
\end{verbatim}
in
let anonStruct =
  <data type definition>({uses, undefined,
    <data type heading>({value, mn □ “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 anonLiterals =
      <data type definition>({empty, undefined,
        <data type heading>({value, mn □ “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 >}
    <operation signature>({operation preamble>({virtual, undefined), c.s<name> □ “Modify”,
      <argument>({undefined, c.s<sort>}), <result>({d.identifier0), empty) | c in choices >
    in
let addMethods =
  <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>({mn □ “Par”, c.s<sort>})}),
      <operation result>({undefined, d.identifier0), empty}, empty),
    <statement list>({empty,
      <return statement>({return body>({expression gen primary>({undefined,
        <operator application>({mn □ “Make”,
          <literal identifier>({c.s<name>}),
        <operator application>({“Make”,
          <if par=c then <variable access>({mn □ “Par”}) else undefined endif
            | c in choices >))))
    in
let addMethodDefs =
  <operation definition>({empty,
    <operation heading>({method, <operation preamble>({virtual, undefined), empty,
      c.s<name> □ “Modify”,
        ITU-T Z.100/Annex F2 (11/2000) 159
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:

```plaintext
let presentOpDef =
<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>(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 >

let realChoice =
<data type definition>(uses, preamble,
<data type heading>(k, n, empty, virt),
<data type specialization>
<type expression>(<identifier>(d.fullQualifier0, nn ◦ “Par”), undefined),
<renaming>(< <rename pair gen operation name>(nn ◦ “Make”, “Make”),
<rename pair gen operation name>(nn ◦ “PresentModify”, “PresentModify”),
<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) 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:

```plaintext
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:

```plaintext
object type Choice-name inherits anonChoice (anonMake = Make,
  anonPresentModify = PresentModify,
  anonPresentExtract = PresentExtract,
  anonChoiceModify = ChoiceModify,
  anonChoiceExtract = ChoiceExtract )
```

adding

```plaintext
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:

```plaintext
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:

```plaintext
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:

```plaintext
PresentExtract ( Choice-name ) -> anonPresent;
```

PresentExtract returns the value associated with the Present field.
2.9.8 Behaviour of Operations

Concrete Syntax

\[
<\text{operation definitions}> =<\text{operation definition}>\mid<\text{external operation definition}>
\]

\[
<\text{textual operation reference}>::<\text{operation kind}><\text{operation signature}>
\]

\[
<\text{external operation definition}>::<\text{operation kind}><\text{operation signature}>
\]

\[
<\text{operation definition}>:::<\text{package use clause}>*<\text{operation heading}><\text{entity in operation}>*
\{<\text{operation body}>|<\text{statement list}>\}
\]

\[
<\text{operation heading}>:::<\text{operation kind}><\text{operation preamble}><\text{qualifier}><\text{operation name}>
\]

\[
<\text{formal operation parameter}>:::[<\text{argument virtuality}>]<\text{parameter kind}><\text{parameters of sort}>
\]

\[
<\text{entity in operation}>=
<\text{data definition}>
\mid<\text{variable definition}>
\mid<\text{exception definition}>
\mid<\text{select definition}>
\]

\[
<\text{operation body}>:::[<\text{on exception}>]<\text{start}>
\{<\text{free action}>|<\text{exception handler}>\}*\]

\[
<\text{operation result}>:::[<\text{variable}><\text{name}>]<\text{sort}>
\]

Conditions on Concrete Syntax

\[
\forall opRef\in<\text{textual operation reference}>::
(opRef<s><\text{operation heading}>.s<\text{formal operation parameter}><\text{seq}=empty\lor
opRef<s><\text{operation heading}>.s<\text{operation result}>=undefined)\Rightarrow
(\forall opRef1\in<\text{textual operation reference}>:
opRef1\neq opRef\land opRef1.parentAS0=opRef1.parentAS0\Rightarrow opRef1.name_0\neq opRef.name_0)
\]

\[
\forall opDef\in<\text{external operation definition}>::
(opDef<s><\text{operation heading}>.s<\text{formal operation parameter}><\text{seq}=empty\lor
opDef<s><\text{operation heading}>.s<\text{operation result}>=undefined)\Rightarrow
(\forall opDef1\in<\text{external operation definition}>:
opDef1\neq opDef\land opDef1.parentAS0=opDef1.parentAS0\Rightarrow opDef1.name_0\neq opDef.name_0)
\]

<\text{formal operation parameters}> and <\text{operation result}> in <\text{textual operation reference}> and <\text{external operation definition}> may be omitted if there is no other <\text{textual operation reference}> or <\text{external operation definition}>, respectively, within the same sort which has the same name.

\[
\forall od\in<\text{operation definition}>::\exists os\in<\text{operation signature}>:
od.parentAS0=os.parentAS0\land od.name_0=os.name_0\land isCompatibleOperationAndSignature_0(od, os)
\]

For each <\text{operation definition}> there must exist an <\text{operation signature}> in the same scope unit having the same <\text{operation name}>, positionally having the same <\text{argument sort}>s and <\text{parameter kind}>'s as specified in the <\text{formal operation parameters}> (if present) and having the same <\text{result sort}> as specified in <\text{operation result}> (if present).
∀ os ∈ ⟨operation signature⟩ : ∃ ode ∈ ⟨operation definition⟩:  
\( \text{od.parentAS0} = \text{os.parentAS0} \land \text{od.name}_0 = \text{os.name}_0 \land \text{isCompatibleOperationAndSignature}_0(\text{od}, \text{os}) \)

For each ⟨operation signature⟩ at most one corresponding ⟨operation definition⟩ can be given.

∀ bs ∈ ⟨operation body⟩ ∃ ⟨statement⟩: \( \text{parentAS0ofKind}(\text{bs}, \langle \text{operation definition} \rangle) \) \# undefined \( \Rightarrow \)  
\( (\nexists \langle \text{imperative expression} \rangle : \text{isAncestorAS0}(\text{bs, id}) \land \)  
\( (\nexists \langle \text{identifier} \rangle : \text{id.idKind} \in \{\text{synonym, procedure}\} \land \text{isAncestorAS0}(\text{bs, id}) \Rightarrow \)  
\( \text{isDefinedInProcEntityDefinition}(\text{id, id.idKind}, \text{parentAS0ofKind}(\text{bs}, \langle \text{operation definition} \rangle))) \)

⟨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**

\( \text{od} = \langle \text{operation definition} \rangle (*, \langle \text{operation heading} \rangle (\text{kind}, \text{preamble}, *, \text{name, params, result, raises}), *, *) \)

\( \text{provided} \rightarrow \exists \text{os} \in \langle \text{operation signature} \rangle : \text{isCompatibleOperationAndSignature}_0(\text{od}, \text{os}) \land \text{kind} = \text{method} \)

\( = 2 \Rightarrow \text{od} = \langle \text{operation definition} \rangle (*, \langle \text{operation heading} \rangle (\text{kind}, \text{preamble}, *, \text{name, params, result, raises}), *, *) \)

\( \text{provided} \rightarrow \exists \text{os} \in \langle \text{operation signature} \rangle : \text{isCompatibleOperationAndSignature}_0(\text{od}, \text{os}) \land \text{kind} = \text{operator} \)

\( = 2 \Rightarrow \text{od} = \langle \text{operation definition} \rangle (*, \langle \text{operation heading} \rangle (\text{kind}, \text{preamble}, *, \text{name, params, result, raises}), *, *) \)

For every ⟨operation definition⟩ which does not have a corresponding ⟨operation signature⟩, an ⟨operation signature⟩ is constructed.

let \( \text{nn = newName in} \)

\( \text{od} = \langle \text{operation definition} \rangle (\text{use, \langle operation heading \rangle (kind, *, *, \text{name, *}, \text{params, \langle result \rangle (var, sort), raises, entities, body)}) \)

\( \text{provided} \od\text{.operatorProcedureName} = \text{undefined} \)

\( = 5 \Rightarrow \)

\( \od\)

and\n
\( \od\text{.getEntities} \Rightarrow \langle \text{procedure definition} \rangle (\text{use, \langle procedure heading \rangle (undefined, undefined, nn, <=, undefined, undefined, \} (\text{if kind = method then}

\( \langle \text{if formal procedure parameter} \rangle (\text{iout, \langle parameters of sort \rangle (\langle \text{thisname >, parentAS0ofKind}(\text{od, \langle data type definition \rangle }, \text{idIdentifier}_0) \rangle) \)

\( \text{else } <= \text{endif} \rangle \)

\( \langle \text{if formal procedure parameter} \rangle (\text{p.s-<parameter kind>, p.s-<parameters of sort>}) \mid \text{p in params} \rangle, \)

\( \langle \text{procedure result} \rangle (\text{var, sort), raises, entities, makeProcedureBody(body)}) \rangle \)

and\n
\( \od\text{.operatorProcedureName} = \text{nn} \)

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
respective 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 determines if an <operation definition> and an <operation
signature> are compatible.

isCompatibleOperationAndSignature(od: <operation definition>, os: <operation signature>): BOOLEAN
= def
let seq1 = od.operationFormalparameterList0 in
let seq2 = os.operationSignatureParameterList0 in
(od.s-<operation heading>.s-<operation result> ≠ undefined ⇒
  isSameResult(od.s-<operation heading>.s-<operation result>, os.s-<result>) ∧
  seq1.length = seq2.length ∧
  (∀ i ∈ 1..seq1.length: isSameSort(seq1[i].parentAS0.s-<sort>, seq2[i].s-<sort>) ∧
    seq1[i].parentAS0.s-<parameter kind>= seq2[i].s-<parameter kind>))
endlet

Get the list of formal parameters of an operation definition.

operationFormalparameterList0(od: <operation definition>): <name>* = 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(i: <identifier>): ENTITYKIND0 = 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, TYPEDEFINITION0).kind0
| <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, TYPEDEFINITION0).kind0
| <procedure preamble> => remote procedure
| <raises> => exception
<channel endpoint> =>
  if getEntityDefinition(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(i, signal) ≠ undefined then signal
  elsif getEntityDefinition(i, signallist) ≠ undefined then signallist
  elsif getEntityDefinition(i, timer) ≠ undefined then timer
  elseif getEntityDefinition(i, remote procedure) ≠ undefined then remote procedure
  elseif getEntityDefinition(i, remote variable) ≠ undefined then remote variable
  else interface endelse

<remote procedure call body> => remote procedure

<import expression> => variable

<entry point> => state

<raise body> => exception

<create body> =>
  if getEntityDefinition(i, agent) ≠ undefined then agent
  else agent type
  endif

<procedure call body> => procedure

<output body> => signal

<via path> =>
  if getEntityDefinition(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, TYPEDEFINITION0).kind0

a = <actual context parameter> =>
  take({f ∈ <formal context parameter>: isContextParameterCorresponded(a, f ) }).entityKind0

<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>
  ∪ <syntype definition gen syntype> ∪ <sort constraint> ∪ <syntype definition gen syntype> => sort

<method application> => method

<operator application> => operator

<indexed primary> ∪ <field primary> ∪ <actual context parameter> ∪ <expression gen primary> =>
  if getEntityDefinition(i, variable) ≠ undefined then variable
  elseif getEntityDefinition(i, synonym) ≠ undefined then synonym
  else literal
  endif

<stimulus> => variable

<exception stimulus> => variable

<assignment> => variable

<indexed variable> => variable
\[ \text{case } b \text{ of} \]

\[ \text{| } \text{<operation body>(onexc, start, actions)} \Rightarrow \text{<procedure body>(onexc, start, actions)} \]

\[ \text{| } \text{<statement list>(*, *) } \Rightarrow \text{<compound statement>}(b) \]

\[ \text{otherwise } \]

\[ \text{undefined} \]

\[ \text{endcase} \]

### 2.9.9 Additional Data Definition Constructs

#### 2.9.9.1 Name Class

**Abstract Syntax**

\[
\text{Nameclass-literal} :: \text{Regular-Expression}* \\
\text{Regular-Expression} :: \text{Partial-regular-expression}* \\
\text{Partial-regular-expression} :: \text{ANY} \\
\text{Regular-element [Regular-multiplicity]} \\
\text{Regular-multiplicity} = \text{AGENTNAT} \\
\text{ATLEASTONCE} \\
\text{ANY} \\
\text{Regular-element} = \text{Nameclass-literal} \mid \text{TOKEN} \mid \text{Regular-interval} \\
\text{Regular-interval} :: \text{TOKEN} \text{ TOKEN} \\
\]

**Concrete Syntax**

\[
\text{<name class operation>} :: \text{<operation<name> <regular expression>} \\
\text{<name class literal>} :: \text{<regular expression>} \\
\text{<regular expression>} :: \text{<or partial regular expression>}+ \\
\text{<or partial regular expression>} :: \text{<partial regular expression>}+ \\
\text{<partial regular expression>} :: \text{<regular element> [\text{<Natural<name> | <plus sign> | <asterisk>}]} \\
\text{<regular element>} = \text{<regular expression>} \\
\mid \text{<character string>} \\
\mid \text{<regular interval>}
\]

**Conditions on Concrete Syntax**

\[
\forall \text{os} \in \text{<operation signature>} : \text{os.s- s-implicit } \in \text{<name class operation>} \Rightarrow \\
\text{os. parentAS0} \in \text{<operator list>} \land \text{os.s=<argument>-seq = empty}
\]

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 is used to determine if a name is in the specified regular expression.

\[
\text{isInRegularExpr}(\text{name}, \text{regExpr: <or partial regular expression>}) : \text{BOOLEAN} = \begin{cases} 
\text{False} & \text{if } \text{regExpr} = \text{empty} \\
\text{isInPartialRegularExpr}(\text{name}, \text{regExpr.head.s:<partial regular expression>-seq}) \lor \text{isInRegularExpr}(\text{name}, \text{regExpr.tail}) & \text{else}
\end{cases}
\]

The function isInPartialRegularExpression is true if the name matches the partial regular expression.

\[
\text{isInPartialRegularExpression}(\text{name}, \text{regExpr: <partial regular expression>-seq}) : \text{BOOLEAN} = \begin{cases} 
\text{name} = \text{empty} & \text{if } \text{regExpr} = \text{empty}
\end{cases}
\]
else $\exists \, \text{name}_1, \text{name}_2 \in <\text{name}>: \text{name}_1 \cap \text{name}_2 = \text{name}$ \wedge
isInPartialRegularExpression(name_2, regExpr.tail) \wedge

\text{case regExpr.head.s-implicit of}
| <\text{name}> => regularExpressionRepeated(name_2, value(regExpr.head.s-implicit), regExpr.head.s-\text{<regular element>})
| <\text{plus sign}> => $\exists \, \text{name}_{21}, \text{name}_{22} \in <\text{name}>: \text{name}_{21} \cap \text{name}_{22} = \text{name}_2$ \wedge
  regularElementMatches(name_{21}, regExpr.head.s-\text{<regular element>}) \wedge
  asteriskMatches(name_{22}, regExpr.head.s-\text{<regular element>})
| <\text{asterisk}> => asteriskMatches(name_2, regExpr.head.s-\text{<regular element>})

endcase
endif

The function \text{regularExpressionRepeated} is true if name matches exactly count times to regExpr.

regularExpressionRepeated(name: <\text{name}>, count: \text{Nat}, regExpr: <\text{regular element}>): \text{BOOLEAN} = \text{def}
if count = 0 then
  name = empty
elseif count = 1 then
  regularElementMatches (name, regExpr)
else
  ($\exists \, \text{name}_1, \text{name}_2 \in <\text{name}>: \text{name}_1 \cap \text{name}_2 = \text{name}$ \wedge
  regularElementMatches (name_1, regExpr) \wedge
  regularExpressionRepeated(name_2, count - 1, regExpr))

The function \text{asteriskMatches} is true if the regular expression repeated any number of times matches the name.

asteriskMatches(name: <\text{name}>, regExpr: <\text{regular element}>)=\text{def}
if name = empty then
  True
else
  ($\exists \, \text{name}_1, \text{name}_2 \in <\text{name}>: \text{name}_1 \cap \text{name}_2 = \text{name}$ \wedge
  regularElementMatches (name_1, regExpr) \wedge
  asteriskMatches(name_2, regExpr))

The function \text{regularElementMatches} is true if regExpr matches name.

regularElementMatches(name: <\text{name}>, regExpr: <\text{regular element}>) =\text{def}
if regExpr \in <\text{regular expression}> then
  isInRegularExpr(name, regExpr.s-<\text{or partial regular expression}>-seq)
else regExpr \in <\text{character string}> then
  name = regExpr
else
  name.length=1 \wedge regExpr.s-<\text{character string}>[1] \leq name[1] \wedge
  name[1] \leq regExpr.s-<\text{character string}>[1]
endif

The function determines if the specified two <name class literal>s are the same.

\text{isSameNameClassLiteral}(ncl1:<\text{name class literal}>, ncl2:<\text{name class literal}>): \text{BOOLEAN} = \text{def}
\forall n <\text{name}>:
  (isInRegularExpr(n, ncl1) \Rightarrow isInRegularExpr(n, ncl2)) \wedge
  (isInRegularExpr(n, ncl2) \Rightarrow isInRegularExpr(n, ncl1))

\text{isSameNameClassOperation}(ncl1:<\text{name class operation}>, ncl2:<\text{name class operation}>): \text{BOOLEAN} = \text{def}
ncl1.s-<\text{name}>= ncl2.s-<\text{name}>

2.9.9.2 Name Class Mapping

Concrete Syntax

<spelling term>::= <operation><name>
Transformations

\[
\text{op} = \text{<operator application>(identifier, parameters)}
\]

\text{provided } \exists \text{ op } \in \text{<name class operation> :}
\[
\text{resolutionByContext}_0(\text{identifier}) = \text{op}
\]
\[\land \neg \text{ identifier.implicitSignaturesAdded}\]

= \text{=} \Rightarrow \text{op}

\text{and}

\text{let origop } = \text{resolutionByContext}_0(\text{identifier}) \text{ in}

\text{let ops } = \text{take(\{op } \in \text{<operations> : origop.parentAS0.parentAS0 } = \text{op1\})} \text{ in}

\text{ops.} \text{<operation definitions>-seq} = \text{=} \Rightarrow \text{ops.} \text{<operation definitions>-seq} \text{ \land}

< \text{replaceInSyntaxTree(<spelling term>(identifier.s-<name>), <literal>(identifier.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

\[
\text{<visibility>} = \text{public} \mid \text{protected} \mid \text{private}
\]

#### Conditions on Concrete Syntax

\[
\forall d \in \text{<data type definition>} : d.specification_0 \neq \text{undefined } \Rightarrow
\]
\[
(\forall con \in \text{<data type constructor>} : \text{con.surroundingScopeUnit}_0 = d \Rightarrow \text{con.s-<visibility>} = \text{undefined})
\]

<visibility> must not precede a <literal list>, <structure definition>, or <choice definition> in a <data type definition> containing <data type specialization>.

\[
\forall os \in \text{<operation signature>} : \text{os.virtuality}_0 \in \{\text{redefined, finalized}\} \Rightarrow \text{os.visibility}_0 = \text{undefined}
\]

<visibility> must not be used in an <operation signature> that redefines an inherited operation signature.

#### Transformations

\[
\text{<operation preamble>(virt, public) provided vis } \neq \text{ undefined } = 1 \Rightarrow \text{<operation preamble>(virt, undefined)}
\]

\[
\text{<literal list>(public, sigs) provided vis } \neq \text{ undefined } = 1 \Rightarrow \text{<literal list>(undefined, sigs)}
\]

\[
\text{<structure definition>(public, fields) provided vis } \neq \text{ undefined } = 1 \Rightarrow \text{<structure definition>(undefined, fields)}
\]
<fields of sort>(public, fields, sort) provided vis ≠ undefined => <fields of sort>(undefined, fields, sort)

<choice definition>(public, fields) provided vis ≠ undefined => <choice definition>(undefined, fields)

<choice of sort>(public, fields, sort) provided vis ≠ undefined => <choice of sort>(undefined, fields, sort)

If a <literal signature> or <operation signature> contains the keyword public in <visibility>, this is derived syntax for a signature having no protection.

### Auxiliary Functions

The function $isPrivate_0$ determines if a <literal signature> or an <operation signature> is private.

$$isPrivate_0(s: \text{<literal signature> } \cup \text{<operation signature> }): BOOLEAN =_{\text{def}} (s.\text{visibility}_0 = \text{private})$$

The function $isPublic_0$ determines if a <literal signature> or an <operation signature> is public.

$$isPublic_0(s:\text{<literal signature> } \cup \text{<operation signature> }): BOOLEAN =_{\text{def}} (s.\text{visibility}_0 \neq \text{private}) \land (s.\text{visibility}_0 \neq \text{protected})$$

#### 2.9.9.4 Syntypes

### Abstract Syntax

$$Syntype-definition ::=$ Syntype-name
Parent-sort-identifier
Range-condition

Parent-sort-identifier =Sort-identifier

### Concrete Syntax

$$<\text{syntype}> = <\text{syntype}<\text{identifier}>$$

$$<\text{syntype definition}> ::=
<\text{package use clause}>* \{ <\text{syntype definition gen syntype}> | <\text{syntype definition gen type preamble}> \}$$

$$<\text{syntype definition gen syntype}> ::=<\text{syntype}<\text{name}> <\text{parent sort identifier}>
[<\text{default initialization}>] [<\text{constraint}>]$$

$$<\text{syntype definition gen type preamble}> ::=<\text{type preamble}> <\text{data type heading}> [<\text{data type specialization}>]
[<\text{data type definition body}>] <\text{constraint}>$$

$$<\text{parent sort identifier}> = <\text{sort}>$$

### Transformations

Let $nn = \text{newName in}$

$$<\text{syntype definition}(\text{uses},<\text{syntype definition gen type preamble}(\text{preamble},<\text{data type heading}(\text{kind, name, params, vconstr}, \text{spec, body, constr}))) > =_{8=}
<\text{syntype definition}(\text{uses},<\text{syntype definition gen syntype}(\text{name, <identifier>(<>}, nn), \text{undefined, constr}),<\text{data type definition}(\text{uses, preamble},<\text{data type heading}(\text{kind, nn, params, vconstr},\text{spec, body})>)$$

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

∀sd∈<syntype definition>: sd.s-implicit.s<constraint>∈<range condition>⇒

(let rc = sd.s-implicit.s<constraint> in

  (∀sym∈<less than sign>: isAncestorAS0(rc, sym)⇒isDefinedSym(sd, "\<")∧
   (∀sym∈<greater than sign>: isAncestorAS0(rc, sym)⇒isDefinedSym(sd, ">")∧
   (∀sym∈<less than or equals sign>: isAncestorAS0(rc, sym)⇒isDefinedSym(sd, "\<=")∧
   (∀sym∈<greater than or equals sign>: isAncestorAS0(rc, sym)⇒isDefinedSym(sd, ">="))

)endlet)

The symbol “<” must only be used in the concrete syntax of the <range condition> if that symbol has been defined with an <operation signature>: "<" (P, P) → <<package Predefined>>Boolean; where P is the sort of the syntype, and similarly for the symbols ("<=", ">", "<>", respectively).

∀sd∈<syntype definition>: sd.s-implicit.s<constraint>∈<range condition>⇒

∀cr∈<closed range>: isAncestorAS0(sd.s-implicit.s<range condition>,cr)⇒isDefinedSym(sd, "\<=")

A <closed range> must only be used if the symbol "<=" is defined with an <operation signature>: "\<=" (P, P) → <<package Predefined>>Boolean; where P is the sort of the syntype.

∀sd∈<syntype definition>: ∀rc∈<range condition>: ∀ce∈<constant expression>:

  isAncestorAS0(rc, ce)∧rc.surroundingScopeUnit=sd⇒isSameSort(ce.staticSort₀, sd.identifier₀)
A <constant expression> in a <range condition> must have the same sort as the sort of the syntype.

∀sd∈<syntype definition>: ∀sc∈<size constraint>:
sc=sd.s-implicit.s-<constraint>⇒isDefinedSym(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(const)) |
| 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 isPredefSort0 is used to determine the predefined sorts.

isPredefSort(s: <sort> : BOOLEAN =def |
getEntityDefinition(s, sort)=undefined<s.s=<name>∈PREDEFINEDSORT0 |

The function isDefinedSym0 is used to determine if the given symbol is defined and the each parameter’s sort is the same as that of the specified syntype.

isDefinedSym(sd:<syntype definition>, sym: SYMBOL0 : BOOLEAN =def |
(let dtd=sd.derivedDataType in |
if sym∈{"","","","","",""} then |
(3ll<literal list>: ll.surroundingScopeUnit0 = dtd)∧ |
(3os<operation signature>: (os.surroundingScopeUnit0=ll)∧ |
(let fpl=os.operationSignatureParameterList0 in |
os.entityName0=sym∧ |
isPredefSort(os.s=<result>)∧ os.s=<result>.s=<name>="Boolean"∧ |
fd.length = 2∧ |
getEntityDefinition(fpl[1].s=<formal parameter>.s=<sort>, sort)=sd∧ |
getEntityDefinition(fpl[2].s=<formal parameter>.s=<sort>, sort)=sd |
endlet)) |
else // sym={"Length"} |
(3os<operation signature>: os.surroundingScopeUnit0=dtd)∧ |
(let fpl=os.operationSignatureParameterList0 in |
os.name0="Length"∧ |
isPredefSort(os.s=<result>)∧ os.s=<result>.s=<name>="Natural"∧ |
endlet))}
2.9.9.6 Synonym Definition

Concrete Syntax

\(<\text{synonym definition}> :: <\text{synonym definition item}>+\)

\(<\text{synonym definition item}> = <\text{internal synonym definition item}> | <\text{external synonym definition item}>\)

\(<\text{internal synonym definition item}> :: <\text{synonym name}> [<\text{sort}>] <\text{constant expression}>\)

\(<\text{external synonym definition item}> :: <\text{synonym name}> <\text{predefined sort}>\)

Conditions on Concrete Syntax

\(\forall \text{syno} \in <\text{internal synonym definition item}>: \neg \text{isContainedInConsExp}_0(\text{syno}, \text{syno}.s_-<\text{constant expression}>\))

The <constant expression> must not refer to the synonym defined by the <synonym definition> either directly or indirectly (via another synonym).

\(\forall \text{sd}\in <\text{internal synonym definition item}>: \text{sd}.s_-<\text{sort}> \neq \text{undefined} \Rightarrow \exists \text{s} \in <\text{sort}>: \text{s} \in \text{sd}.s_-<\text{constant expression}>.\text{staticSortSet}_0.\text{isSameSort}_0(s, sd.s_-<\text{sort}>\)

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.

\(\forall \text{sd}\in <\text{internal synonym definition item}>: |
\text{sd}.s_-<\text{constant expression}>.\text{staticSortSet}_0| > 1 \Rightarrow \text{sd}.s_-<\text{sort}> \neq \text{undefined}\)

If the sort of the <constant expression> cannot be uniquely determined, then a sort must be specified in the <synonym definition>.

Auxiliary Functions

The function isContainedInConsExp0 is used to determine if a <constant expression> refer to the synonym defined by the enclosing <synonym definition> either directly or indirectly.

\(\text{isContainedInConsExp}_0(\text{def}, <\text{internal synonym definition item}>, \text{exp}, <\text{constant expression}>): \text{BOOLEAN} =\def\
\exists \text{synoId} \in <\text{synonym}>: \text{isAncestor}_0(\text{exp}, \text{synoId}) \land
(\text{def} = \text{getEntityDefinition}_0(\text{synoId}, \text{synonym}) \lor
\text{isContainedInConsExp}_0(\text{def}, \text{getEntityDefinition}_0(\text{synoId}, \text{synonym}).s_-<\text{constant expression}>))\)

2.9.10 Expression

2.9.10.1 Expression

Abstract Syntax

\[
\begin{array}{c|c|c|c}
\text{Expression} & = & \text{Constant-expression} & \text{Active-expression} \\
\text{Constant-expression} & = & \text{Literal} & \text{Conditional-expression} & \text{Equality-expression}
\end{array}
\]
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

\[
\text{<expression>} = \begin{cases}
\text{<create expression>}
| \text{<value returning procedure call>}
| \text{<range check expression>}
| \text{<binary expression>}
| \text{<equality expression>}
| \text{<expression gen primary>}
\end{cases}
\]

\[
\text{<simple expression>} = \begin{cases}
\text{<constant expression>}
\end{cases}
\]

\[
\text{<constant expression>} = \begin{cases}
\text{<constant expression>}
\end{cases}
\]

\[
\text{<binary expression>} :: \begin{cases}
\text{<expression>}
| \text{<implies sign> | or | xor | and}
| \text{<greater than sign> | <greater than or equals sign> | <less than sign> | <less than or equals sign> | in}
| \text{<plus sign> | <hyphen> | <concatenation sign> | <asterisk> | <solidus> | mod | rem}
\end{cases}
\]

\[
\text{<expression gen primary>} :: \begin{cases}
\text{<hyphen> | not | <primary>}
\end{cases}
\]

\[
\text{<primary>} = \begin{cases}
\text{<operator application>}
| \text{<literal>}
| \text{<expression>}
| \text{<conditional expression>}
| \text{<spelling term>}
| \text{<extended primary>}
| \text{<active primary>}
| \text{<synonym>}
\end{cases}
\]

\[
\text{<active primary>} = \begin{cases}
\text{<variable access> | <imperative expression>}
\end{cases}
\]

\[
\text{<imperative expression>} = \begin{cases}
\text{<now expression>}
| \text{<import expression>}
| \text{<pid expression>}
| \text{<timer active expression>}
| \text{<any expression>}
| \text{<state expression>}
\end{cases}
\]

Conditions on Concrete Syntax

\[
\forall \text{expr} <\text{expression}>: \begin{cases}
(expr.parentAS0 \in <\text{default initialization}> \cup <\text{timer default initialization}> \cup <\text{field default initialization}> \cup
\end{cases}
\]
A constant expression must not contain an active primary.

\[ \forall expr : expr.parentAS0 = \langle \text{number of instances} \rangle \cup \langle \text{named number} \rangle \cup \langle \text{transition option} \rangle \Rightarrow isSimpleExpression(expr) \]

A simple expression must contain only literals, operators, and methods defined within the package Predefined, as defined in Annex D.

Transformations

- \[ <\text{binary expression}(x, \langle \text{infix operation name} \rangle, y) = \langle \text{operator application}("\langle \text{infix operation name} \rangle", x, y) \rangle \]
- \[ <\text{binary expression}(x, \langle \text{infix operation name} \rangle, y) = \langle \text{operator application}("\langle \text{infix operation name} \rangle", x, y) \rangle \]
- \[ <\text{binary expression}(x, \langle \text{monadic operation name} \rangle, y) = \langle \text{operator application}("\langle \text{monadic operation name} \rangle", x) \rangle \]

An expression of the form:

\[ <\text{expression}> <\text{infix operation name}> <\text{expression}> \]

is derived syntax for:

\[ <\text{quotation mark}> <\text{infix operation name}> <\text{quotation mark}> ( <\text{expression}> , <\text{expression}> ) \]

where \[ <\text{quotation mark}> <\text{infix operation name}> <\text{quotation mark}> \] represents an Operation-name.

Similarly,

\[ <\text{monadic operation name}> <\text{expression}> \]

is derived syntax for:

\[ <\text{quotation mark}> <\text{monadic operation name}> <\text{quotation mark}> ( <\text{expression}> ) \]

where \[ <\text{quotation mark}> <\text{monadic operation name}> <\text{quotation mark}> \] represents an Operation-name.

Auxiliary Functions

Get the one of the possible static sort for an expression:

\[ staticSort_{\text{def}}(expr; <\text{expression}>); : <\text{sort}> = \text{def} \]

\[ \text{take}(expr, staticSortSet_{\text{def}}) \]

Determine if an expression is a constant expression. A constant expression must not contain an active primary.

\[ isConstantExpression_{\text{def}}(expr; <\text{expression}>); : \text{BOOLEAN} = \text{def} \]

\[ \neg \exists ap \in \langle \text{active primary} \rangle : isAncestorAS0(expr, ap) \]
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.

\[
\text{isSimpleExpression}(\text{expr}) : \text{BOOLEAN} \equiv (\forall \text{id} \in \text{<identifier>}: \text{isAncestorAS}(\text{expr}, \text{id}) \land \text{id} \cdot \text{idKind} \in \{\text{literal, operator, method}\} \Rightarrow \text{getEntityDefinition}(\text{id}, \text{id} \cdot \text{idKind}) \in \text{PREDEFINEDDEFINITION}) \land \\
\text{isConstantExpression}(\text{expr})
\]

### 2.9.10.2 Literal

**Abstract Syntax**

\[
\text{Literal} :: \text{Literal-identifier}
\]

**Concrete Syntax**

\[
\text{<literal>} = \text{<literal identifier>}
\]

\[
\text{<literal identifier>} :: \text{<qualifier> <literal name>}
\]

**Mapping to Abstract Syntax**

\[
| \text{<literal identifier>}(<\text{qual}, \text{name}) \Rightarrow \text{mk-Literal-identifier}(\text{Mapping(qual)}, \text{Mapping(name)})
\]

### 2.9.10.3 Synonym

**Concrete Syntax**

\[
\text{<synonym>} :: \text{<synonym><identifier>}
\]

**Transformations**

\[
\text{<synonym>}(<\text{ident}) \Rightarrow \text{ident.refersto0} \in \text{<internal synonym definition item>}
\]

\[
\text{provided ident.refersto0} \equiv \text{<constant expression>}
\]

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**

\[
\text{<extended primary>} = \\
| \text{<indexed primary>} \\
| \text{<field primary>} \\
| \text{<composite primary>}
\]

\[
\text{<indexed primary>} :: \text{<primary>} [\text{<actual parameter>}]^+ \\
\text{<field primary>} :: [\text{<primary>}] <\text{field name}> \\
\text{<field name>} = <\text{name}> \\
\text{<composite primary>} :: <\text{qualifier}> [\text{<actual parameter>}]^+
\]
Transformations

An <indexed primary> is derived concrete syntax for

<primary> Extract ( <actual parameter list> )

The abstract syntax is determined from this concrete expression according to 12.2.1.

A <field primary> is derived concrete syntax for

<primary> 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.

When the <field primary> has the form <field name>, this is derived syntax for

this ! <field name>

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<sexpression>.staticSortSet0 in
let set2 = `ee.s2<sexpression>.staticSortSet0 in
3s1 ∈ set1:3s2 ∈ set2:isSortCompatible0(s1, s2)→ isSortCompatible0(s2, s1)
endlet)

An <equality expression> is legal concrete syntax only if the sort of one of its operand is sort compatible to the sort of the other operand.
Transformations

\[ (x, \text{not equality expression}, y) = 8 \Rightarrow \text{expression gen primary}(\text{not}, (x, \text{equality expression}, y)) \]

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

\[
\text{Conditional-expression} :: \text{Boolean-expression} \\
\text{Consequence-expression} \\
\text{Alternative-expression}
\]

Consequence-expression = Expression

Alternative-expression = Expression

Conditions on Abstract Syntax

\[ \forall c \in \text{Conditional-expression}:
(c.s-\text{Consequence-expression}.\text{staticSort} = c.s-\text{Alternative-expression}.\text{staticSort}) \land
(c.s-\text{Boolean-expression}.\text{staticSort} = \text{mk-Identifier(Predefined, Boolean)}) \]

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

\[
<\text{conditional expression}> :: <\text{Boolean-expression}> <\text{consequence expression}> <\text{alternative expression}>
\]

<consequence expression> = <expression>

<alternative expression> = <expression>

Conditions on Concrete Syntax

\[ \forall \text{e} \in <\text{conditional expression}>:
\begin{align*}
\text{let } & \text{set}1 = \text{e.s-}\text{<consequence expression>.staticSortSet}_0, \text{ in} \\
\text{let } & \text{set}2 = \text{e.s-}\text{<alternative expression>.staticSortSet}_0, \text{ in} \\
|\text{set}1| = 1 \land |\text{set}2| = 1 \land \text{isSameSort(\text{set}1.\text{take}, \text{set}2.\text{take})}
\end{align*}
\]

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

\[
\text{Operation-application} :: \text{Operation-identifier [ Expression ]*}
\]
Conditions on Abstract Syntax

\[ \forall oa \in \text{Operation-application:} \]
\[
\text{let } os = \text{getEntityDefinition}(oa, \text{operation}) \text{ in}
\]
\[
\text{isActualAndFormalParameterMatched}(oa.s-\text{Expression-seq}, os.\text{formalParameterSortList})
\]
endlet

The Operation-identifier in the Operation-application must be visible. Each Expression in the list of Expressions 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

\[
<\text{operation application}> = <\text{operator application}> | <\text{method application}>
\]
\[
<\text{operator application}> :: <\text{operation identifier}> [<\text{actual parameter}>]*
\]
\[
<\text{method application}> :: <\text{primary}> <\text{operation identifier}> [<\text{actual parameter}>]*
\]

Conditions on Concrete Syntax

\[ \forall \text{methodApp} \in <\text{method application}>:
\]
\[
\text{getEntityDefinition}(\text{methodApp.s-<identifier>, method}) \neq \text{undefined}
\]

A <method application> is legal concrete syntax only if operation identifier represents a method.

Transformations

\[
<\text{method application}>(\text{prim}, \text{ident}, \text{params}) =8=>
\]
\[
<\text{operator application}>(\text{ident}, <\text{prim} > \land \text{params})
\]

The concrete syntax form:

\[
<\text{expression}> <\text{full stop}> <\text{operation identifier}> [<\text{actual parameters}>]
\]

is derived concrete syntax for:

\[
<\text{operation identifier}> \text{ new-actual-parameters}
\]

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.

\[
\text{actualParameterListOfOpId}(id: \text{Operation-identifier}): [\text{Expression}]^* = \text{def}
\]
\[
\text{case } id.\text{parentAS1} \text{ of}
\]
\[
| \text{Open-range} => <id.\text{parentAS1}.s-Constant-expression>
| \text{Operation-application} => <exp | exp in id.\text{parentAS1}.s-Expression-seq>
\]
endcase

2.9.10.8 Range Check Expression

Abstract Syntax

\[
\text{Range-check-expression} :: \text{Range-condition Expression}
\]
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(rcExpr.s-<expression>.staticSort, s)
endlet

The sort of <operand2> must be the same as the sort identified by <sort identifier> or <sort>.

Transformations

<range check expression>(i = <identifier>(*,*))
=⇒
  if i.refersto0 ∈ <syntype definition> then
    <range check expression>(<range check expression gen identifier>(i, i.refersto0.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>
=⇒
  <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 = 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> [<remote variable<identifier>]

Conditions on Concrete Syntax

∀ea∈<variables of sort gen name>:
  ea.s-<identifier> ≠ undefined ⇒ ea.parentAS0.parentAS0.isExported0

<exported as> may only be used for a variable with exported in its <variable definition>.

∀d∈<agent definition>∪<agent type definition> :
  ¬(∃v1,v2∈<variables of sort gen name>: v1 ≠ v2 ∧ v1.s-<identifier> = v2.s-<identifier> ∧
  v1.parentAS0.parentAS0∈<variable definition> ∧
  v2.parentAS0.parentAS0∈<variable definition> ∧
  v1.surroundingScopeUnit0 = d ∧ v2.surroundingScopeUnit0 = d)

Two exported variables in an agent cannot mention the same <remote variable<identifier>.

Transformations

< <variable definition>(exp, <v> ∩ rest) > provided rest ≠ empty =1=⇒
  < <variable definition>(exp, <v>), <variable definition>(exp, rest) >

< <variables of sort> (<v> ∩ rest, sort, expr) > provided rest ≠ 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

∀va∈<variable access>: va.s-implicit = this ⇒
  (parentAS0ofKind(va, <operation definition>) ≠ undefined ∧
  parentAS0ofKind(va, <operation definition>).kind0=method)

this must only occur in method definitions.
Transformations

\( va = \langle \text{variable access}\rangle(\text{this}) \)

\( =8\Rightarrow \langle \text{variable access}\rangle(\text{parentAS0ofKind}(va, \langle \text{operation definition}\rangle).s<\text{operation heading}>.s<\text{formal operation parameter}>.\text{head}.s<\text{parameters of sort}>.s<\text{name}>.\text{head}) \)

A \( \langle \text{variable access}\rangle \) using the keyword this is replaced by the anonymous name introduced as the name of the leading parameter in \( \langle \text{arguments}\rangle \) according to 12.1.8.

Mapping to Abstract Syntax

\[ \langle \text{variable access}\rangle(\text{identifier}) \Rightarrow \text{mk-Variable-identifier}(\text{Mapping}(\text{identifier})) \]

### 2.9.10.11 Assignment and Assignment Attempt

#### Abstract Syntax

<table>
<thead>
<tr>
<th>Assignment</th>
<th>::</th>
<th>Variable-identifier Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment-attempt</td>
<td>::</td>
<td>Variable-identifier Expression</td>
</tr>
</tbody>
</table>

#### Conditions on Abstract Syntax

\[ \forall a \in \text{Assignment} : \exists d \in \text{Variable-definition} : \]

\[ (d = \text{getEntityDefinition}(a.s-\text{Variable-identifier}.\text{variable})) \land \]

\[ \text{isCompatibleTo}(a.s-\text{Expression}.\text{staticSort}_1, d.s-\text{Sort-reference-identifier}) \]

In an Assignment, the sort of the Expression must be sort compatible to the sort of the Variable-identifier.

\[ \forall aa \in \text{Assignment-attempt} : \exists d \in \text{Variable-definition} : \]

\[ (d = \text{getEntityDefinition}(aa.s-\text{Variable-identifier}.\text{variable})) \land \]

\[ \text{isCompatibleTo}(d.s-\text{Sort-reference-identifier}, aa.s-\text{Expression}.\text{staticSort}_1) \]

In an Assignment-attempt, the sort of the Variable-identifier must be sort compatible to the sort of the Expression.

#### Concrete Syntax

\[ \langle \text{assignment} \rangle :: \langle \text{variable} \rangle \langle \text{expression} \rangle \]

\[ \langle \text{variable} \rangle = \langle \text{variable} \rangle <\text{identifier}> | <\text{indexed variable}> | <\text{field variable}> \]

#### Mapping to Abstract Syntax

\[ \langle \text{assignment} \rangle(\text{var,expr}) \]

\[ =\Rightarrow \text{if isAttempt}(\text{var.staticType,expr.staticType}) \text{ then} \]

\[ \text{mk-Assignment-attempt}(\text{var,expr}) \]

\[ \text{else} \]

\[ \text{mk-Assignment}(\text{var,expr}) \]

\[ \text{endif} \]

### 2.9.10.12 Extended Variable

#### Concrete Syntax

\[ \langle \text{indexed variable} \rangle :: \langle \text{variable} \rangle [\langle \text{actual parameter} \rangle]^+ \]

\[ \langle \text{field variable} \rangle :: \langle \text{variable} \rangle <\text{field name}> \]
Transformations

\[<\text{assignment}>(<\text{indexed variable}>(\text{var}, \text{params}), \text{expr}) \Rightarrow<\text{assignment}>(\text{var}, <\text{method application}>(\text{var}, <\text{identifier}>(<>", \text{"Modify"}), \text{expr}))\]

<indexed variable> is derived concrete syntax for

\[<\text{variable} > <\text{is assigned sign}> <\text{variable} > <\text{full stop}> \text{Modify (expressionlist)}\]

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>.

\[<\text{assignment}>(<\text{field variable}>(\text{var}, \text{fieldname}), \text{expr}) \Rightarrow<\text{assignment}>(\text{var}, <\text{method application}>(\text{var}, \text{modifyExtractName(fieldname, "Modify")}, \text{expr}))\]

The concrete syntax form:

\[<\text{variable} > <\text{exclamation mark}> <\text{field name}> <\text{is assigned sign}> <\text{expression}>\]

is derived concrete syntax for:

\[<\text{variable} > <\text{full stop}> \text{field-modify-operation-name ( <expression> )}\]

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

\[
\text{modifyExtractName}(name: <\text{identifier}>, suffix: \text{TOKEN}):<\text{identifier}> =\text{def}\\
\text{mk}<\text{identifier}>(name.s-<\text{qualifier}>, name.s-<\text{name}> + suffix)
\]

2.9.10.13 Default Initialization

Concrete Syntax

\[<\text{default initialization}> :: [<\text{virtuality}>] [<\text{constant expression}>]\]

Conditions on Concrete Syntax

\[
\forall \text{def} \in <\text{data type definition}>: \neg \exists d_1, d_2 \in <\text{default initialization}>: \text{def} \neq \text{def} \\
\forall sd \in <\text{syntype definition}>: (sd.s-\text{implicit.s}<\text{default initialization}>\neq\text{undefined}) \Rightarrow \\
\neg \exists d \in <\text{default initialization}>: \text{isDefinedIn}_0(d, sd.derivedDateType_0)
\]

A <data type definition> or <syntype definition> must contain at most one <default initialization>.

\[
\forall \text{init} \in <\text{default initialization}>: \\
\text{init.s}-<\text{constant expression}> = \text{undefined} \Rightarrow \text{init.virtuality} \in \{\text{redefined, finalized}\}
\]

The <constant expression> may only be omitted if <virtuality> is redefined or finalized.

Transformations

\[
<\text{variable definition}>(\text{undefined}, < <\text{variables of sort}>(<\text{variables of sort gen name}>(\text{name},*), \text{sort}, \text{undefined}>) >) \Rightarrow
\]

provided

\[
\text{getEntityDefinition}(\text{sort, sort}). \text{getDefaultInitialization} \neq \text{undefined}
\]

\[=\Rightarrow
\]

\[
<\text{variable definition}>(\text{undefined}, < <\text{variables of sort}>(<\text{variables of sort gen name}>(\text{name},*), \text{sort}, \text{getEntityDefinition}(\text{sort, sort}). \text{getDefaultInitialization} >)
\]
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 = def
ease 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(d, type)}) in
  if init = undefined then
    let parentSort = type.derivedDateType in
    if parentSort = undefined then
      If type.s=<data type heading>.s=<data type kind> = object v type.isPidSort then
        <operator application>(<operation identifier>(<>", Null"), <>)
      else
        undefined
    else
      getParent(getDefaultInitialization(parentsort))
    endif
  endif
  else
    init
  endif
endlet
```

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>
=⇒
  <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.s←<sort>, undefined)) in
  items=getEntities(exp)
=⇒ < dcl >^ items
)

let nn = newName in
  exp = <import expression>(ident, *)
  provided parentAS0ofKind(exp, <transition gen action statement>) ≠ undefined
  ∧ exp.parentAS0.parentAS0 ∉ <assignment>
=⇒
  <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.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


Concrete Syntax


Transformations

let nn = newName in

exp = <create expression>(body = <create body>(target, params))

provided parentAS0ofKind(exp,<action>) ≠ undefined

=>

<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

=>

<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**

\[
\forall t \in \text{Timer-active-expression}:
\]

```plain
let d = getEntityDefinition(t.s-Timer-identifier, timer) in
t.s-Expression.length = d.s-Sort-reference-identifier. length ∧
(\forall i \in 1..t.s-Expression. length : isCompatibleTo(t.s-Expression[i].staticSort, 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**

\[
\forall \text{exp} \in \text{<any expression>}: \text{isContainingElements}_0(<\text{exp.<sort>})
\]

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(s, sort): BOOLEAN = def
let d = getEntityDefinition(s, sort) in
(de PREDEFINEDSORT0)\n(\exists cons <data type constructor>: cons.surroundingScopeUnit0 = d &
(cons <structure definition> \& <choice definition>)\n(\forall s <sort>: sort in cons.fieldSortList0 \Rightarrow isContainingElements(s, sort))\n(d.specialization0 \neq undefined)\n(isContainingElements(d.specialization0, s <type expression> . baseType0)\n(\forall acp <actual context parameter>: acp in d . actualContextParameterList0 \& acp.idKind0 = sort \Rightarrow isContainingElements(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

\(\forall exp <\text{continuous expression}> : exp.parentAS0 <\text{continuous signal}> \Rightarrow
\neg\exists procCall <value returning procedure call> : isAncestorAS0(exp, procCall)\)

\(\forall exp <\text{provided expression}> : exp . parentAS0 <\text{input part}> \Rightarrow
\neg\exists procCall <value returning procedure call> : isAncestorAS0 (exp, procCall)\)

A <value returning procedure call> must not occur in the <Boolean expression> of a <continuous signal> or <enabling condition>.

\(\forall procId <\text{identifier}> : procId . parentAS0 <\text{value returning procedure call}> \Rightarrow
getEntityDefinition(procId, procedure), s <procedure heading>, s <procedure result> \neq undefined\)

The <procedure identifier> in a <value returning procedure call> must identify a procedure having a <procedure result>.

\(\forall procCall <\text{procedure call body}> :\)

\(procCall <value returning procedure call> \land procCall.s-this \neq undefined \Rightarrow
getEntityDefinition(procId, procedure) = parentAS0ofKind(procCall, <procedure definitions>)\)

If this is used, <procedure identifier> must denote an enclosing procedure.
Transformations

let $nn = \text{newName}$ in
$p = \langle \text{value returning procedure call} \rangle (\langle \text{procedure call body} \rangle (id, params) )$

provided $\text{parentAS0ofKind}(id, \text{refersto}_0, <\text{agent type definition}>) \neq$
$\text{parentAS0ofKind}(p, <\text{agent type definition}>)$

=8=>
let $par = \text{parentAS0ofKind}(p, <\text{agent type definition}>)$ in
\langle \text{value returning procedure call} \rangle (\langle \text{procedure call body} \rangle (\langle \text{identifier} \rangle (par.fullQualifier\_0 \cap \langle \text{path item} \rangle (par.entityKind\_0, par.entityName\_0), nn),
params))
endlet

and // add the new definition
let $defs = \text{parentAS0ofKind}(p, <\text{agent type definition}>).s->\text{agent structure}.s->\text{entity in agent}->\text{seq in}$
$defs = \Rightarrow$ $\text{def in}$
\langle \text{procedure definition} \rangle (empty,
\langle \text{procedure heading} \rangle (\langle \text{procedure preamble} \rangle (\text{undefined, undefined),
empty, $nn$, empty, $\text{undefined, undefined, empty, undefined, empty, empty),
empty,
\langle \text{procedure body} \rangle (\text{undefined, undefined, empty}))
endlet

If the $<\text{procedure identifier}>$ 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 $<\text{procedure identifier}>$ is replaced by the identifier of the
specialized procedure.

let $nn = \text{newName}$ in
$p = \langle \text{value returning procedure call} \rangle (\langle \text{remote procedure call body} \rangle (id, params, constrs, onexc)
=8=>
\langle \text{value returning procedure call} \rangle (\langle \text{procedure call body} \rangle (nn, <>))
and // add the new definition
let $defs = \text{parentAS0ofKind}(p, <\text{agent type definition}>).s->\text{agent structure}.s->\text{entity in agent}->\text{seq in}$
$defs = \Rightarrow$ $\text{def in}$
\langle \text{procedure definition} \rangle (empty,
\langle \text{procedure heading} \rangle (\langle \text{procedure preamble} \rangle (\text{undefined, undefined),
empty, $nn$, empty, $\text{undefined, undefined, empty, undefined, empty, empty),
empty,
\langle \text{procedure body} \rangle (\text{undefined, undefined}))
endlet

When the $<\text{value returning procedure call} \rangle$ contains a $<\text{remote procedure call body} >$, the following procedure with an
anonymous name referred to as RPCcall is implicitly defined. RPCsort is the $<\text{sort}>$ in $<\text{procedure result}>$ of the
procedure definition denoted by the $<\text{procedure identifier} >$.

\begin{verbatim}
procedure RPCcall -> RPCsort;
    start;
    return call <remote procedure call body>;
endprocedure;
\end{verbatim}

NOTE – This transformation is not again applied to the implicit procedure definition.
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 \(<\text{qualifier}>\)s in every \(<\text{identifier}>\) 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) \(<\text{macro definition}>\)s and \(<\text{macro call}>\)s (6.2) are identified lexically and \(<\text{macro call}>\)s are expanded;
   b) \(<\text{macro definition}>\)s are removed (also in \(<\text{package definition}>\)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 \(<\text{choice definition}>\).

3) \(<\text{task}>\)s, \(<\text{task area}>\)s, and \(<\text{statement list}>\)s are transformed as defined in 11.13.1 and 11.14.

4) Definition references are replaced by \(<\text{referenced definition}>\)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 \(<\text{free action}>\)s. Those generated \(<\text{free action}>\)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 \(<\text{free action}>\).
   - \(<\text{operation definition}>\)s are transformed into procedures having anonymous names and having the result as \(<\text{procedure result}>\). The resulting \(<\text{procedure definition}>\) is moved to the enclosing scope unit.

6) The package Predefined is included in the \(<\text{sdl specification}>\).

7) Transformation of generic system (see clause 13) and external data (see 12.2.3):
   - identifiers in \(<\text{simple expression}>\)s contained in the \(<\text{sdl specification}>\) are bound to definitions. During this binding, only \(<\text{data definition}>\)s defined in the predefined package Predefined and \(<\text{external synonym definition}>\)s are considered (that is, all other \(<\text{data definition}>\)s are ignored);
   - \(<\text{external synonym}>\)s are replaced by \(<\text{synonym definition}>\)s and informal text in transition options is replaced by \(<\text{range condition}>\). How this is done is not defined by SDL.
   - \(<\text{simple expression}>\)s are evaluated and \(<\text{select definition}>\)s, \(<\text{option area}>\)s, \(<\text{transition option}>\)s and \(<\text{transition option area}>\)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
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
customOfExp0, 19, 31; defined at, 31
customOfIdentifier0, 19; defined at, 19
customParameterAtleastDefinition0, 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
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
techniqueName0, 10, 14, 15, 16, 18, 19, 27, 35, 37, 38, 68, 102, 124, 172, 189; defined at, 10
techniqueName1, 7; defined at, 7
textPointSet1, 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
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, 51
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; defined at, 9
isExported0, 78, 89, 96, 181; defined at, 79
isInOpenRange1, 9; defined at, 9
isInPartialRegularExpression, 167, 168; defined at, 167
isInRegularExp0, 16, 167, 168; defined at, 167
isMentionedInDefSelection0, 18; defined at, 18
isObjectSort0, 142, 143, 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, 144, 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, 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
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
isSatisfyExtendedPrimaryCond0, 25; defined at, 25
isSatisfyIndexVariableCondition0, 22; defined at, 22
isSatisfyMethodAppCondition0, 25, 27; defined at, 27
isSatisfyOpAppCondition0, 25, 27; defined at, 27
isSatisfyPrimaryCondition0, 23, 24, 25, 27; defined at, 24
isSatisfyRemoteProcCallBodyCondition0, 23, 24; defined at, 24
isSatisfyRangeCheckCondition0, 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
virtualTypeAtleastDefinition0, 65, 66, 68; defined at, 67

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, 5
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

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 constraint 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
<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 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
<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>, 21, 29, 171, 175, 176; 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>, 90, 92, 98, 100, 125, 126; defined at, 125
<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, 54, 59, 146; 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>, 11, 31, 34, 39, 59, 73, 89, 90, 93, 95, 103; defined at, 89
<remote variable reject>, 94, 103, 104; defined at, 103
<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
<select 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; 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
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, 49, 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-definition, 44, 80; defined at, 108
Composite-state-definition, 5, 6, 7, 8, 34, 36, 41, 43, 44, 45, 77, 79, 80, 111, 113, 114; defined at, 44
Composite-state-definition, 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
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, 32
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, 13
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
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, 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
<table>
<thead>
<tr>
<th>Series</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Organization of the work of ITU-T</td>
</tr>
<tr>
<td>B</td>
<td>Means of expression: definitions, symbols, classification</td>
</tr>
<tr>
<td>C</td>
<td>General telecommunication statistics</td>
</tr>
<tr>
<td>D</td>
<td>General tariff principles</td>
</tr>
<tr>
<td>E</td>
<td>Overall network operation, telephone service, service operation and human factors</td>
</tr>
<tr>
<td>F</td>
<td>Non-telephone telecommunication services</td>
</tr>
<tr>
<td>G</td>
<td>Transmission systems and media, digital systems and networks</td>
</tr>
<tr>
<td>H</td>
<td>Audiovisual and multimedia systems</td>
</tr>
<tr>
<td>I</td>
<td>Integrated services digital network</td>
</tr>
<tr>
<td>J</td>
<td>Cable networks and transmission of television, sound programme and other multimedia signals</td>
</tr>
<tr>
<td>K</td>
<td>Protection against interference</td>
</tr>
<tr>
<td>L</td>
<td>Construction, installation and protection of cables and other elements of outside plant</td>
</tr>
<tr>
<td>M</td>
<td>TMN and network maintenance: international transmission systems, telephone circuits, telegraphy, facsimile and leased circuits</td>
</tr>
<tr>
<td>N</td>
<td>Maintenance: international sound programme and television transmission circuits</td>
</tr>
<tr>
<td>O</td>
<td>Specifications of measuring equipment</td>
</tr>
<tr>
<td>P</td>
<td>Telephone transmission quality, telephone installations, local line networks</td>
</tr>
<tr>
<td>Q</td>
<td>Switching and signalling</td>
</tr>
<tr>
<td>R</td>
<td>Telegraph transmission</td>
</tr>
<tr>
<td>S</td>
<td>Telegraph services terminal equipment</td>
</tr>
<tr>
<td>T</td>
<td>Terminals for telematic services</td>
</tr>
<tr>
<td>U</td>
<td>Telegraph switching</td>
</tr>
<tr>
<td>V</td>
<td>Data communication over the telephone network</td>
</tr>
<tr>
<td>X</td>
<td>Data networks and open system communications</td>
</tr>
<tr>
<td>Y</td>
<td>Global information infrastructure and Internet protocol aspects</td>
</tr>
<tr>
<td>Z</td>
<td>Languages and general software aspects for telecommunication systems</td>
</tr>
</tbody>
</table>